Approximate computing in numerical linear algebra: algorithms, analysis, and applications

Théo Mary
Chargé de recherche, CNRS

Mémoire d'habilitation à diriger des recherches

présenté et soutenu publiquement le 7 Octobre 2025
devant un jury composé de :

Rapporteurs:		
Hartwig Anzt	Professeur	Technical University of Munich (Allemagne)
Julien Langou	Professeur	University of Colorado, Denver (Etats-Unis)
Jean-Michel Muller	Directeur de recherche	CNRS, LIP (France)
Examinateurs:		
Paolo Bientinesi	Professeur	Umeå University (Suède)
Stef Graillat	Professeur	Sorbonne Université, LIP6 (France)
Xiaoye Sherry Li	Senior scientist	Lawrence Berkeley National Lab (Etats-Unis)
Valeria Simoncini	Professeur	Università di Bologna (Italie)

To the memory of Nick Higham

Remerciements (Acknowledgments)

Lorsqu'on a eu la chance de travailler avec un ensemble de personnes aussi vaste et varié que j'ai pu l'avoir au cours de ces dix dernières années, écrire le chapitre de remerciements devient un défi de grande taille!

Ce manuscrit d'habilitation porte sur mes travaux postérieurs à ma thèse de doctorat. C'est pourtant par remercier mes directeurs de thèse—Patrick Amestoy, Alfredo Buttari, et Jean-Yves L'Excellent—que je me dois de commencer. Vous m'avez accompagné dans mes premiers pas en recherche, formé le chercheur que je suis devenu, et avez été et resterez pour toujours des modèles de rôle pour moi.

Si mes débuts dans la recherche ont eu lieu à l'IRIT, c'est au LIP6 que je me suis épanoui. Je n'utilise pas le verbe épanouir au hasard: le LIP6, et en particulier mon équipe PEQUAN, présente un environnement de travail que je trouve sain, humain, joyeux et stimulant, et qui a été un facteur clé dans cet épanouissement. Je remercie en particulier Fabienne Jézéquel et Stef Graillat: vous êtes les piliers de cette équipe et vous avez su y créer cet environnement. Merci de m'avoir ouvert toutes les portes à mon arrivée. Merci à Pierre Jolivet qui a réjoint l'équipe plus récemment: nous partageons déjà une multitude de projets enthousiasmants. Merci à Mehdi El Arar, avec qui j'ai le plaisir de travailler depuis quelques années: je suis ravi de ton arrivée dans l'équipe. Merci à Dominique Béréziat: nous ne travaillons pas ensemble, mais j'apprécie les pauses déjeuner en ta compagnie! En dehors de PEQUAN, merci à nos chers voisins de couloir de POLSYS et en particulier à Jérémy Berthomieu, avec qui j'ai le plaisir de découvrir le monde du calcul formel. Merci tout autant à toutes les personnes des services administratifs, financiers et informatiques du LIP6 pour leur impressionante efficacité. Un grand merci en particulier à Noura El Habchi, pour ta grande patience malgré mes aller/retour incessants entre Lyon et Paris! Merci à Hélène Petridis et Konstantin Kabassanov pour votre aide et réactivité dans les derniers préparatifs pour le workshop suivant l'HDR.

Avec l'IRIT et le LIP6, mon troisième laboratoire de cœur est le LIP, où j'ai la chance d'être accueilli lors de mes "séjours" à Lyon. Merci en particulier à l'équipe ROMA qui m'accueille dans ses locaux, mais aussi aux équipes PASCALINE, OCKHAM et AVALON. Cette pluralité thématique autour du calcul donne au LIP un positionnement unique et pour moi l'occasion de travailler avec de formidables collègues: merci à Grégoire Pichon, à Nicolas Brisebarre et Claude-Pierre Jeannerod, à Rémi Gribonval et Elisa Riccietti, à Thierry Gautier et Pierre-Etienne Polet. J'espère que cette liste s'allongera.

Je souhaite également remercier l'équipe MUMPS et dire toute l'admiration que j'ai pour ce projet et ma fierté de pouvoir y contribuer. Plus qu'un projet, MUMPS est une famille qui m'accompagne depuis le début. Merci de nouveau à Patrick et Jean-Yves, et merci à Chiara Puglisi, dont le rôle dans cette famille est bien plus important qu'elle ne se l'accorde!

Au travers de divers projets nationaux et des opportunités de recherche qui se sont présentées

à moi, j'ai eu la chance de pouvoir collaborer avec de nombreux autres chercheurs en France, qui ont énormément enrichi mon expérience. J'ai beaucoup appris et je continue tous les jours de découvrir de nouvelles choses grâce à eux. Merci donc à Emmanuel Agullo et Luc Giraud, à Frédéric Nataf et Pierre-Henri Tournier, à Marc Baboulin, à Oguz Kaya, à Silviu Filip, à Clément Pernet et à Florent Lopez. La recherche n'est pas l'apanage du monde académique, loin de là, et je souhaite remercier tous mes collègues du secteur industriel et/ou applicatif. Travailler avec eux est un plaisir, l'opportunité d'en apprendre davantage sur les problèmes que nos algorithmes résolvent, et de ce fait une source incessante de motivation. Merci en tout premier lieu à Olivier Boiteau d'EDF, pour son soutien de longue date à MUMPS; j'espère que nous avons encore de belles saisons devant nous. Merci à Stéphane Operto, Laure Combe, Alain Miniussi et le reste de l'équipe Géoazur. Merci à Florian Faucher et Hélène Barucq d'Inria Makutu, à Ani Anciaux-Sédrakian et Thomas Guignon de l'IFPEN, et à Cédric Content et Emeric Martin de l'ONERA.

Derrière la plupart de ces belles collaborations, se cache le travail acharné d'un doctorant ou postdoctorant. Dans l'ordre chronologique, je remercie Bastien Vieublé, Matthieu Gerest, Roméo Molina, Théo Beuzeville, Matthieu Robeyns, Sébastien Dubois, Dimitri Lesnoff, Hugo Dorfsman, Antoine Jego, Yongseok Jang, Tom Caruso, Karmijn Hoogveld, Giuseppe Carrino, Ouassel El Habti, Alexandre Tabouret, Erik Fabrizzi, et Liam Burke. Les succès présentés dans ce manuscrit sont d'abord et surtout les vôtres. Merci pour votre énergie et pour votre motivation—c'est grâce à elles que je garde les miennes.

Enfin, je remercie ma famille et mes amis. Grazie ad Elisa, per il tuo costante sostegno, senza il quale niente di tutto questo sarebbe stato possibile. Grazie per tutte le cose belle che abbiamo fatto insieme, scientifiche e non, di cui la più bella è certamente la nostra piccola Vittoria.

Let me switch to English to thank my non-Francophone colleagues!

I thank my habilitation committee—Hartwig Anzt, Paolo Bientinesi, Stef Graillat, Julien Langou, Sherry Li, Jean-Michel Muller, and Valeria Simoncini—for having agreed to serve and for their many questions and comments during the defense. I especially thank the reviewers—Hartwig, Julien, and Jean-Michel—for their detailed reports and comments that helped me to improve this manuscript. Finally, special thanks to Jean-Michel for his encouragement in expanding this manuscript into a book: somehow, he managed to convince me to undertake this—surely unreasonable—project!

Thank you to all my colleagues and co-authors, with whom I have had the pleasure of working in the past—recently or less so—and to whom I owe many of the accomplishments presented in this manuscript. Special thanks to Massimiliano Fasi and Mantas Mikaitis, with whom we share ongoing exciting projects.

Thank you also to all the researchers with whom I have crossed paths, at conferences or elsewhere, and who have enriched my knowledge and ideas with their own. At the risk of forgetting someone, and with the bias of recency, let me thank in particular Hussam Al Daas, Hartwig Anzt, Grey Ballard, Erin Carson, Desmond Higham, Julien Langou, Sherry Li, Andrew Lister, Hatem Ltaief, Gunnar Martinsson, Jean-Michel Muller, Takeshi Ogita, Philipp Petersen, Jennifer Scott, Françoise Tisseur, George Turkiyyah, and Rio Yokota.

Thank you to the many people who participated in the workshop on approximate computing that followed the defense, and especially to the speakers who gave excellent talks. Thank you also to all those who would have liked to participate but were unable to. Together, we form a

wonderful nascent community at the intersection of several synergistic fields—numerical linear algebra, high performance computing, computer arithmetic, etc. I look forward to seeing what this community will become and the interesting and impactful science it is certain to produce.

I conclude these words by dedicating this manuscript to Nick Higham, the person who had the most impact not only on my career, but also the very way that I approach our profession. My relationship with Nick started when I was his postdoc for two years, and we remained close colleagues until his untimely passing. It is hard to put into words just how big a void Nick has left in my life and, I believe, in our community. There rarely goes a day where I do not think of him: sometimes it's an idea I think he would have appreciated, more often it's a difficult problem I wish I could ask him about. However, Nick has also left us with an incredible legacy: his many books and papers of course, but perhaps more importantly, all the people whom he mentored or inspired. I therefore strongly believe that it is our duty, as a community, to carry on Nick's legacy, by striving for that same level of rigor, clarity, and creativity that Nick somehow managed to combine in everything he undertook.

Contents

Chapter 1

Introduction

1.1 Approximate computing: context and challenges

At the dawn of the exascale computing era, high performance computing (HPC) faces challenges of unprecedented scale and complexity. Computational science applications now require the solution of problems of very large dimensions (millions or even billions of unknowns), and with increasingly complex requirements (storage, speed, energy, and accuracy constraints). Modern supercomputer architectures have also become increasingly complex to use efficiently, because of their large size, the increasing cost of communications relative to that of computations, and the heterogeneity of their computing units (CPUs, accelerators such as GPUs, specialized hardware such as tensor cores or TPUs). Moreover, because of the increasing power consumption of these computers, energy efficiency has also become a primary concern.

My research aims at confronting these challenges by developing approximate computing methods. Introducing numerical approximations offers significant benefits indeed: they can make the computations less expensive, more efficient, and/or easier to perform. First, compressing the size of the data naturally reduces the storage requirements, the data movement, and the volume of communications. Such compression can be achieved by using more compact arithmetic representations (such as lower precision floating-point numbers) or algebraic ones (such as low-rank matrices). Second, approximate computations are faster too, thanks to a reduced number of floating-point operations (flops) and/or a higher speed on modern hardware; in particular, lower precision arithmetics can be orders of magnitude faster on specialized GPU accelerators.

Despite their promising potential, there remains a wide gap that separates approximate computing from standard high performance computing, which must be bridged in order for the use of approximations to reach widespread acceptance. Indeed, one could imagine the use of approximations to become standard, rather than optional, in numerical libraries such as BLAS or LAPACK. Before that becomes possible, three hurdles must be overcome. Indeed, an approximate algorithm can only be considered successful if the following three conditions are *all* met:

1. Its accuracy can be provably and practically controlled.
2. Its performance on parallel computers is usually higher, and always at least equivalent, than that of its exact (non approximate) counterpart.
3. The first two conditions hold for any input data, or at least for a very wide range of data corresponding to real-life applications.

Objective 1: provable and practical control of the accuracy

The first and most immediate challenge of approximate computing is naturally to control the accuracy of the computation, since approximations inevitably introduce errors. For example, half precision arithmetic only provides between three and four accurate digits instead of sixteen with double precision arithmetic. The experimental side of approximate computing has been developed at a faster rate than its theoretical side, leading to many proposed approaches that lack an error analysis to support them. As a result, many of today's approximate algorithms are not able to guarantee stability, and many of them actually do encounter non-obvious numerical issues that can have potentially disastrous consequences. Numerical approximations therefore cannot be used blindly for general purpose scientific computing; the first objective of my research is to develop control and correction mechanisms that ensure that the size of the errors introduced remains bounded and that make sure that the result still achieves sufficient accuracy.

Objective 2: high performance on parallel computers

While approximate methods are less expensive than their exact counterparts, they are also often much less efficient at taking advantage of modern parallel computers. There are several reasons for this counterintuitive fact. First, while approximations can reduce both the operation and communication costs, they typically reduce the former by a much larger factor than the latter; hence approximate computations are even more communication-bound than the usual. Second, approximations are often unpredictable, and therefore lead to unbalanced workloads that are much harder to schedule in parallel environments. As a result, most of today's approximate algorithms only achieve a small fraction of their true performance potential. The second objective of my research is to close the gap between the theoretical potential and actual performance of approximate algorithms, by developing implementations that maintain a high arithmetic intensity and parallel efficiency.

Objective 3: robustly handling a wide range of data from real-life applications

The final challenge lies in the fact that the success of approximations, both in terms of accuracy and of performance, strongly depends on the data on which they are used, and hence on the application at hand. Indeed, approximate algorithms are usually more specialized and therefore less robust than their exact counterparts: they may achieve higher performance but only for a specific, narrower range of input data, and will fail if used on data outside this range. An obvious example is that low-rank approximations are only meaningful if the data is actually low-rank; a less obvious one is that mixed precision iterative methods will only converge for matrices that are sufficiently well conditioned. Moreover, due to the lack of error analysis, the robustness of many algorithms has not been quantified precisely: that is, we often do not know whether a given algorithm will work on a given problem in advance. This a major shortcoming because when a failure occurs, at best, it is detected and the algorithm must be run again more precisely—leading to a major overhead cost—and, at worst, it goes undetected and may have insidious effects later in the computation. The final objective of my research is therefore to quantify and maximize the robustness of approximate algorithms, so as to expand the range of problems that can be handled.

Unfortunately, performance, accuracy, and robustness are conflicting objectives: it is easy to improve one by worsening the others. In fact, it is worth pointing out that achieving two out of the three objectives is straightforward: with an unlimited performance budget, any level of accuracy can eventually be obtained for any input data; conversely we may always be able to find some synthetic input data for which high accuracy and high performance can both be achieved; and naturally, high performance can be achieved regardless of the input if the result does not need to be accurate. The challenge, therefore, is to develop algorithms that meet all three objectives simultaneously.

1.2 Summary of my contributions

To develop algorithms that simultaneously achieve performance, accuracy, and robustness, approximate computing requires a deep understanding of the complete solution process, from the mathematical foundations behind the design of the algorithms, to their practical high performance implementation on parallel computers and their evaluation on real-life applications. A numerical analyst cannot design a new approximate method without appreciating the HPC implications, at the risk of the method being impractical; an HPC programmer cannot develop a new approximate method without grasping the subtleties of error analysis, at the risk of the method being mathematically unsound and unsafe; and neither of them can evaluate the true potential of their method without a realistic application where the tradeoff between approximations and performance can be meaningfully assessed.

Approximate computing therefore requires a multidisciplinary effort across several fields from mathematics and computer science, spanning numerical analysis, linear algebra, high performance computing, computer arithmetic, and all application fields of computational science. My research aims at bringing together these fields by designing, developing, and analyzing approximate algorithms that are, at the same time, able to achieve high performance on parallel supercomputers, mathematically sound with provable accuracy guarantees, and sufficiently robust to handle a wide range of realistic applications.

Thus, an important strand of my research has dealt with developing error analyses to achieve a better understanding of the practical behavior of numerical algorithms. This includes the development of probabilistic analyses to obtain more realistic bounds than the traditional worst-case ones [1, 2, 3, 4, 38]; analyses specific to mixed precision specialized hardware such as GPU tensor cores [5, 6, 7]; and the analysis of some widespread approximate algorithms such as low-rank [8, 9, 39] or mixed precision methods [10, 11, 12, 40], to determine under which conditions they behave stably.

Another equally important strand of my research has dealt with the design of new algorithms to accelerate a wide range of linear algebra computations, and their efficient software implementation on parallel computers. In particular, I am actively involved in the development of the open-source sparse direct solver MUMPS [13], and especially in its use of approximations such as block low-rank [13, 14] and mixed precision methods [15, 16, 41] to reduce its memory and time costs. Besides sparse direct solvers, I have also worked on mixed precision sparse iterative solvers [17, 18, 47], summation methods [19, 20], randomized and/or mixed precision low-rank approximations [16, 21, 22, 33], low complexity rank-structured matrix factorizations [23, 24, 25, 26, 42], and dense linear algebra kernels on GPU accelerators, particularly using mixed precision tensor core units [5, 7, 27, 33].

Finally, I strive to evaluate the impact of my research on realistic applications as much as possible. As mentioned, in the context of approximate computing this has become truly indispensable in order to be able to meaningfully assess the actual potential of the methods. For these reasons, I also dedicate a significant amount of effort to work hand-in-hand with application scientists, from the academy or the industry. This has allowed my work to have a strong impact on several computational science applications such as seismic imaging or structural mechanics [28, 34, 35, 29, 47]. Most recently, I have taken an interest in expanding this range of applications outside the realm of linear algebra, in particular to neural network–based machine learning [43, 44, 42], tensor computing [22, 39], and computer algebra [45].

These contributions are the result of close collaborations with many researchers. In particular, a significant part of them were achieved in the context of several PhD theses:

- Bastien Vieublé (University of Toulouse, IRIT, 2019–2022),
- Matthieu Gerest (Sorbonne University, LIP6–EDF, 2020–2023),
- Théo Beuzeville (University of Toulouse, IRIT–Atos, 2021–2024),
- Roméo Molina (Sorbonne University, LIP6, 2021–2024),
- Matthieu Robeyns (University Paris-Saclay, LISN, 2021–2024),
- Dimitri Lesnoff (Sorbonne University, LIP6, 2022–ongoing),
- Hugo Dorfsman (Sorbonne University, LIP6–IFPEN, 2023–ongoing),
- Tom Caruso (Sorbonne University, LIP6–LJLL, 2024–ongoing),
- Karmijn Hoogveld (University of Toulouse, IRIT, 2024–ongoing).

1.3 Chapter organization

The rest of this manuscript presents my most significant research contributions. These cover a variety of topics in high performance computing, numerical linear algebra, and rounding error analysis, and are divided in chapters roughly on an algorithm-by-algorithm basis. Therefore, they are not necessarily meant to be read sequentially; each chapter attempts to be self-contained, and as a result there may be some amount of overlap between some closely related topics.

Chapters 2 to 6 deal with rounding error analysis and with fundamental kernels like matrix multiplication and low-rank approximations. Chapter 2 covers basics on rounding error analysis: floating-point arithmetic, backward error analysis, and some standard error bounds used in the rest of this manuscript. Chapter 3 presents probabilistic analyses and algorithms to better understand, control, and measure the effect of rounding errors. Chapter 4 presents various summation methods and their error analysis, and their use for matrix multiplication. Chapter 5 presents multiword arithmetic, and its application for matrix multiplication in mixed precision and/or over prime finite fields. Finally, Chapter 6 discusses how to compute low-rank approximations with various methods including randomized ones, and making use of other approximation techniques such as low or mixed precision arithmetic.

Chapters 7 to 12 deal with the solution of linear systems, with an emphasis on sparse matrices. Chapter 7 covers the basics on direct methods. Chapter 8 does the same for iterative methods, and discusses their stability and the mixed precision opportunities that they offer. Chapter 9 deals with block low-rank matrices, and how they can be used to reduce the cost of solving linear systems. Chapter 10 presents iterative refinement, with an emphasis on its mixed precision variants. Chapter 11 discusses adaptive precision methods, which dynamically adapt the precisions to the data at hand. Finally, Chapter 12 presents memory accessor approaches that decouple the storage and compute precisions.

Chapter 13 describes butterfly factorizations, and in particular how to quantize them in low precision. Chapters 14 and 15 move beyond the realm of linear algebra. Chapter 14 deals with tensor approximations and their stability. Chapter 15 discusses neural networks, with a focus on the feed forward pass, its error analysis and mixed precision opportunities.

Chapter 16 concludes the manuscript with a discussion of future challenges and opportunities in the field of approximate computing. I also list some concrete research problems that I believe should be investigated in the near future.

1.4 Terminology and notations

Terminology Many different terms have been used in the literature to refer to algorithms that can use several precisions. In this manuscript, we will use "uniform precision" to refer to

algorithms that use the same precision for all their operations, and we will use "mixed precision" to refer to algorithms that can use different precisions for different operations.

Multiword arithmetic (see Chapter 5) is also sometimes called "multiprecision" arithmetic. We will avoid this term due to its frequent confusion with mixed precision algorithms (and for the converse reason, we will also avoid using "multiprecision" to refer to mixed precision algorithms).

Some less frequently used terms include "variable precision", "dynamic precision", "transprecision", etc. We will avoid using these terms as they do not convey any significantly different meaning than mixed precision algorithms. One exception is the "adaptive precision" term, that we will use to refer to a subclass of mixed precision algorithms that dynamically choose their precisions at runtime based on the data appearing in the computation. Adaptive precision algorithms are discussed in Chapter 11.

We will often use the metonymy "precision u" to refer to the "precision *with unit roundoff* u". Note that in the sentence "we perform this step in a lower precision u_{low} than the rest of the steps, which are performed in a higher precision u_{high}", u_{high} is in fact smaller than u_{low}!

Notations As is standard in rounding error analysis, we denote computed quantities with a hat $\widehat{\ }$. We use the ubiquitous constant $\gamma_n = nu/(1-nu)$ (defined in (2.8)) in the error bounds. Whenever we write γ_n there is an implicit assumption that $nu < 1$. When analyzing mixed precision algorithms with several precision parameters, we use a superscript on γ to denote that u carries that superscript as a subscript; thus for example $\gamma_n^{(f)} = nu_f/(1-nu_f)$ and $\gamma_n^{\text{high}} = nu_{\text{high}}/(1-nu_{\text{high}})$.

Throughout the manuscript, we use the unsubscripted norm $\|\cdot\|$ to denote any norm that is consistent (that is, $\|AB\| \le \|A\|\|B\|$). In some analyses where the precise constants in the error bounds are not important, we leave the choice of norm unspecified, with the understanding that all norms are equivalent up to constants. In the analyses where the constants are important, we specify the norm of interest with a subscript: $\|\cdot\|_2$, $\|\cdot\|_F$, and $\|\cdot\|_\infty$ for the spectral norm, the Frobenius norm, and the infinity norm, respectively.

We use without comment matrix inequalities of the form $|A| \le |B|$, where A and B are matrices, to denote the componentwise inequalities $|a_{ij}| \le |b_{ij}|$.

Citations Throughout the manuscript, we use bold citations to distinguish personal publications (for example, **[10]**) from other references (for example, [162]).

Chapter 2

Basics on rounding error analysis

This chapter covers the necessary prerequisites on finite precision computations and their rounding error analysis. Section 2.1 describes floating-point arithmetic and its properties. Section 2.2 discusses some commonly available examples of floating-point arithmetics. Section 2.3 defines backward error analysis and why it is important. Finally, Section 2.4 lists the classical error bounds of fundamental linear algebra kernels.

For a more in-depth discussion, we refer to [162], also informally (but rightfully) known as "the bible of rounding error analysis".

2.1 Floating-point arithmetic

A floating-point number system $\mathbb{F} \subset \mathbb{R}$ is characterized by the following integer parameters:

- the base β, usually equal to 2,
- the precision t, and
- the exponent range $[e_{\min}, e_{\max}]$.

Floating-point numbers $y \in \mathbb{F}$ are then represented under the form

$$y = \pm m\beta^{e-t}. \tag{2.1}$$

This representation can be encoded on a finite number of digits (called bits when $\beta = 2$) as follows.

- The sign $\pm$ is encoded on 1 bit.
- The significand m is an integer satisfying $0 \leq m \leq \beta^t - 1$. Moreover the system is usually assumed to be normalized, meaning that $m \geq \beta^{t-1}$. This amounts to assuming the leading bit of m to be always 1; this *implicit bit* is not stored and m can thus be encoded on $t-1$ bits.
- The exponent $e \in [e_{\min}, e_{\max}]$ is a biased integer encoded on a number of bits that depends on the range; usually $e_{\max} = 2^{s-1} - 1$, $e_{\min} = 1 - e_{\max}$, and $e = e' + e_{\min}$; the unbiased exponent $e' \in [0, 2e_{\max} - 1] = [0, 2^s - 3]$ can thus be encoded on s bits. The exponent range $[e_{\min}, e_{\max}]$ determines the range of representable floating-point numbers

$$|y| \in \left[\beta^{e_{\min}}, \beta^{e_{\max}+1}(1 - \beta^{-t})\right]. \tag{2.2}$$

Table 2.1: Some common floating-point arithmetics.

		Number of bits			Approx. range (2.2)	Unit roundoff $u = 2^{-t}$
		Sign	Exp. (s)	Signif. (t)		
Quadruple	fp128	1	15	112 (+1)	$[10^{-4932}, 10^{+4932}]$	$2^{-113} \approx 1 \times 10^{-34}$
Double	fp64	1	11	52 (+1)	$[10^{-308}, 10^{+308}]$	$2^{-53} \approx 1 \times 10^{-16}$
Single	fp32	1	8	23 (+1)	$[10^{-38}, 10^{+38}]$	$2^{-24} \approx 6 \times 10^{-8}$
Half	fp16	1	5	10 (+1)	$[6 \times 10^{-5}, 65504]$	$2^{-11} \approx 5 \times 10^{-4}$
Half	bfloat16	1	8	7 (+1)	$[10^{-38}, 10^{+38}]$	$2^{-8} \approx 4 \times 10^{-3}$
Quarter	fp8 (e4m3)	1	4	3 (+1)	$[2 \times 10^{-2}, 448]$	$2^{-4} \approx 6 \times 10^{-2}$
Quarter	fp8 (e5m2)	1	5	2 (+1)	$[6 \times 10^{-5}, 57344]$	$2^{-3} \approx 1 \times 10^{-1}$
	fp6 (e2m3)	1	2	3 (+1)	$[1, 7.5]$	$2^{-4} \approx 6 \times 10^{-2}$
	fp6 (e3m2)	1	3	2 (+1)	$[1 \times 10^{-2}, 28]$	$2^{-3} \approx 1 \times 10^{-1}$
	fp4 (e2m1)	1	2	1 (+1)	$[1, 6]$	$2^{-2} = 0.25$

Real numbers $x \in \mathbb{R}$ that are not floating-point numbers must be *rounded*, that is, mapped to some value $\mathrm{fl}(x) \in \mathbb{F}$. Usually we use round-to-nearest, which means $\mathrm{fl}(x)$ is a floating-point number nearest to x. When x is not in the representable range, we say that $\mathrm{fl}(x)$ underflows or overflows. For the numbers in the representable range, we have the following result.

Theorem 2.1 (Theorem 2.2 of [162]). *If $x \in \mathbb{R}$ lies in the range (2.2), then*

$$\mathrm{fl}(x) = x(1+\delta), \quad |\delta| \le u := \frac{1}{2}\beta^{1-t}. \tag{2.3}$$

Theorem 2.1 shows that the relative distance from any real number $x \in \mathbb{R}$ in the representable range to $\mathrm{fl}(x)$ is bounded by the quantity u called the *unit roundoff*; note that $u = 2^{-t}$ in base two. The unit roundoff (and thus the precision t) determines the relative accuracy with which any $x \in \mathbb{R}$ in the representable range can be represented.

In general, the result of operations on floating-point numbers (called floating-point operations or "flops") is no longer a floating-point number. Flops therefore introduce rounding errors; throughout this manuscript we will use the following model.

Model 2.1 (Standard model of floating-point arithmetic).

$$\mathrm{fl}(x \operatorname{op} y) = (x \operatorname{op} y)(1+\delta), \quad |\delta| \le u, \quad \operatorname{op} \in \{+, -, \times, \div\}. \tag{2.4}$$

Model 2.1 assumes that the result of the elementary operations (addition, subtraction, multiplication, and division) is essentially as accurate as the rounded exact result. This model is almost always valid on modern computers, and in particular for IEEE standard arithmetic.

2.2 Some common floating-point arithmetics

Table 2.1 lists some of the most common floating-point arithmetics and their associated representable range and unit roundoff.

We use the abbreviations fp128, fp64, fp32, and fp16 to denote the IEEE quadruple, double, single, and half precision arithmetics, respectively. Double and single precision arithmetics have been supported in hardware for many decades and are the two most widely used arithmetics. Quadruple precision arithmetic is usually not supported in hardware but can be implemented in software; it is quite costly but can be of interest for computations requiring a very high

level of accuracy. Finally, half precision arithmetic was initially introduced as a storage format but hardware support started appearing in the last decade and has been rapidly growing. In addition to these IEEE arithmetics, other non-standard formats have been proposed: bfloat16 is an alternative half precision format that preserves the same range as fp32 at the price of decreased precision compared with fp16.

With the emergence of accelerators specialized for artificial intelligence applications, even lower precision formats are appearing. The Open Compute Project (OCP) [198] specifies two quarter precision (8-bit) formats using a different balance between range and precision, which are notably available on the NVIDIA Hopper architecture. A subsequent OCP standard [226] also introduces two 6-bit formats and one 4-bit formats. These are likely to be available on the forthcoming NVIDIA Blackwell [208] architecture.

This manuscript (and approximate computing in general) is especially interested in using low precision arithmetics (fp32 and lower), due to their significant performance benefits. Representing data in lower precision readily reduces its storage cost and the cost of communicating it across different processors or different levels of the memory hierarchy. Hence memory-bound operations can be accelerated by using low precision data. Moreover, on hardware supporting lower precision operations, compute-bound operations can also be accelerated. For example, fp32 is usually twice faster than fp64. Moreover, some accelerators provide specialized low precision instructions that are much faster than their standard high precision counterpart. In particular, NVIDIA GPUs provide fp16 matrix multiplication units called tensor cores that can be orders of magnitude faster than fp32 arithmetic. Actually, tensor cores are mixed precision fp16/fp32 units and can actually be more accurate than standard fp16 arithmetic; see Section 4.1 for details.

2.3 Backward error analysis

As mentioned above, Model 2.1 assumes that the four elementary operations $\{+, -, \times, \div\}$ have a relative accuracy bounded by the unit roundoff u. However, this does not hold for more complex computations involving a sequence of such elementary operations, because rounding errors accumulate and propagate throughout the computation. The main goal of rounding error analysis is to bound the error incurred by such computations.

Consider an abstract computation $y = f(x)$ where $x \in \mathbb{R}^m$ is the input, $y \in \mathbb{R}^n$ is the output, and the function f implements the computation of interest. In floating-point arithmetic, f will produce a computed $\widehat{y}$ affected by rounding errors. The *forward error* is defined as the distance from $\widehat{y}$ to the exact result y, for a suitable choice of distance: it can be normwise or componentwise, absolute or relative:

$$\begin{cases} \|\widehat{y} - y\| & \text{(normwise absolute)}, \\ \max_i |\widehat{y} - y|_i & \text{(componentwise absolute)}, \\ \frac{\|\widehat{y}-y\|}{\|y\|} & \text{(normwise relative)}, \\ \max_i \frac{|\widehat{y}-y|_i}{|y|_i} & \text{(componentwise relative)}. \end{cases} \tag{2.5}$$

Forward error analysis (the process of measuring and/or bounding forward errors) is however made complicated by the fact that the exact y is in general unknown (making it impossible to measure forward errors) and by the fact that forward errors strongly depend on the input data: for the same computation f, two input x_1 and x_2 can lead to vastly different forward errors when f is ill-conditioned.

This motivates the interest in defining a different error metric that is more practical to measure and easier to interpret. This is achieved by *backward* error analysis, which seeks to express the

computed $\widehat{y}$ as the result of an exact computation on a perturbed input: $\widehat{y} = f(x + \Delta x)$. The backward error is then defined as the size of Δx, again with different possible definitions of size:

$$\begin{cases} \|\Delta x\| & \text{(normwise absolute)}, \\ \max_i |\Delta x|_i & \text{(componentwise absolute)}, \\ \frac{\|\Delta x\|}{\|x\|} & \text{(normwise relative)}, \\ \max_i \frac{|\Delta x|_i}{|x|_i} & \text{(componentwise relative)}. \end{cases} \tag{2.6}$$

Backward error analysis was popularized in the 1960s by James Wilkinson [251] and can be used to evaluate the *stability* of a given algorithm. An algorithm is said to be backward stable if for any input data, it achieves a small backward error, where "small" usually means of order the unit roundoff u. Indeed, x is often only known with an accuracy of order u (because it is the result of a previous computation, or simply because it is stored in floating-point arithmetic), so that $\widehat{y} = f(x + \Delta x)$ with Δx of size u is essentially an ideal result: for all we know, $x + \Delta x$ might be the exact input !

2.4 Standard backward error bounds

In linear algebra, most of the reference standard algorithms are proven to be backward stable. This essentially boils down to the fact that matrix-based computations take the form of sequence of elementary operations such as

$$z = x_1 \times y_1 + x_2 \times y_2 + \dots$$

By repeated application of (2.4) the computed $\widehat{z}$ satisfies

$$\begin{aligned} \widehat{z} &= \mathrm{fl}\left(x_1 \times y_1 + x_2 \times y_2 + \dots\right) \\ &= \big(\mathrm{fl}(x_1 \times y_1) + \mathrm{fl}(x_2 \times y_2)\big)(1 + \delta_3) + \dots \\ &= \big(x_1 y_1 (1 + \delta_1) + x_2 y_2 (1 + \delta_2)\big)(1 + \delta_3) + \dots \\ &= x_1 y_1 (1 + \delta_1)(1 + \delta_3) + x_2 y_2 (1 + \delta_2)(1 + \delta_3) + \dots \end{aligned}$$

which illustrates that the perturbations on the input take the form of products of $(1 + \delta_i)$ terms.

Such perturbations can be bounded by the following fundamental result.

Theorem 2.2 (Lemma 3.1 of [162]). *If $|\delta_i| \le u$ and $\rho_i = \pm 1$ for $i = 1\colon n$, and $nu < 1$, then*

$$\prod_{i=1}^{n} (1 + \delta_i)^{\rho_i} = 1 + \theta_n, \quad |\theta_n| \le \gamma_n, \tag{2.7}$$

where

$$\gamma_n = \frac{nu}{1 - nu}. \tag{2.8}$$

From this result follows the backward stability of inner products and, therefore, that of many inner products–based computations. The next three theorems establish error bounds for inner products, matrix–vector products, and matrix–matrix products, and will be extensively used throughout the manuscript.

Theorem 2.3 (Equation (3.4) of [162]). *Let $x, y \in \mathbb{R}^n$ and $s = x^T y$. For any order of evaluation, the computed $\widehat{s}$ satisfies*

$$\widehat{s} = (x + \Delta x)^T y, \quad |\Delta x| \le \gamma_n |x|. \tag{2.9}$$

Theorem 2.4 (Equation (3.11) of [162]). *Let $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, and $y = Ax$. For any order of evaluation, the computed $\widehat{y}$ satisfies*

$$\widehat{y} = (A + \Delta A)x, \quad |\Delta A| \le \gamma_n |A|. \tag{2.10}$$

Theorem 2.5 (Equation (3.13) of [162]). *Let $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times p}$, and $C = AB$. For any order of evaluation, the computed $\widehat{C}$ satisfies the columnwise backward error bounds*

$$\widehat{c}_j = (A + \Delta A_j)b_j, \quad |\Delta A_j| \le \gamma_n |A|, \tag{2.11}$$

which implies the forward error bound

$$\widehat{C} = C + \Delta C, \quad |\Delta C| \le \gamma_n |A||B|. \tag{2.12}$$

It is worth noting that the constant γ_n in the above bounds can be replaced by simply nu with a more refined analysis of inner products [172]. This also removes the restriction $nu < 1$, although when nu exceeds 1 the bound does not guarantee any correct digit.

Chapter 3

Probabilistic analysis and algorithms

As illustrated in the Section 2.4 of the previous chapter, rounding error bounds of linear algebra computations involving matrices or vectors of size n tend to grow at least linearly with n.

Historically, the role of n in the error bounds has been neglected by numerical analysts. In traditional error analysis, terms only depending on the dimensions of the data are called "constants" [162, sec. 3.2], which reflects the fact that historically n was in practice not large enough for its role in the growth of the error to be deemed significant.

However, this view is no longer valid with the extreme size of matrices (large n) that are nowadays involved with exascale computing, and with the emergence of low precision arithmetics (large unit roundoff u). For example, in half precision arithmetic, error bounds of order nu cannot guarantee even a single correct digit for matrices of order a few thousands, and are anyway inapplicable when $nu > 1$. Worst-case bounds therefore become quite unsatisfactory at large scale and low precisions.

Fortunately, these are worst-case bounds and, in practice, the error often exhibits a much weaker dependence on n. A famous conjecture by Wilkinson [250, p. 318] states that n can be replaced by $\sqrt{n}$ due to statistical effects in the rounding errors: assuming they behave somewhat randomly, positive and negative rounding errors should cancel each other to some extent. This rule of thumb remained formally unproven for 60 years.

In this chapter, we discuss probabilistic error analysis and how it can help us better understand and better control the accumulation of rounding errors. In Section 3.1, we present a new approach to probabilistic backward error analysis that formally proves Wilkinson's conjecture under a well defined model of rounding errors. In Section 3.2, we prove that the use of stochastic rounding enforces rounding errors to follow this model, and can hence significantly improve the accuracy of computations for which the worst-case bound would be attained with standard rounding modes. In Section 3.3, we combine the probabilistic model of rounding errors with a probabilistic model on the data, and prove that the error bounds can be even further reduced when the data has zero mean. Finally, in Section 3.4, we discuss the use of stochastic arithmetic as a tool for numerical validation.

The contributions described in this chapter are based on the following papers: **[1]** for Section 3.1; **[3]** for Section 3.2 (see also our survey **[4]**); **[2]** for Section 3.3; **[38]** for Section 3.4.

3.1 Probabilistic backward error analysis

The key for developing a probabilistic error analysis that is both rigorous and informative is the model that one uses for rounding errors. It must be sufficiently constrained to lead to reduced error bounds, and at the same time sufficiently general to apply to a wide range of

representative computations. Here, we will use the following model which is the result of a series of papers starting with **[1]** and including **[2, 3]**.

Model 3.1. *Let the computation of interest generate rounding errors $\delta_1, \dots, \delta_n$ in that order. The δ_k are random variables such that $\mathbb{E}(\delta_k \mid \delta_1, \dots, \delta_{k-1}) = \mathbb{E}(\delta_k) = 0$.*

Model 3.1 assumes the rounding errors to be zero-mean, mean independent variables. Mean independence is the property that the conditional expectation of δ_k given all previous rounding errors, $\mathbb{E}(\delta_k \mid \delta_1, \dots, \delta_{k-1})$, is equal to its unconditional expectation $\mathbb{E}(\delta_k)$ (which is zero). It is a weaker assumption than independence. Our initial model in **[1]** assumed independence, but was later refined to Model 3.1 in **[2]**.

Under this model, we can prove a probabilistic analogue of Theorem 2.2, a fundamental tool of backward error analysis.

Theorem 3.1 (Theorem 4.6 in **[3]**)**.** *Let $\delta_1, \dots, \delta_n$ be rounding errors satisfying Model 3.1 and define*

$$\widetilde{\gamma}_n(\lambda) = \exp\left(\frac{\lambda\sqrt{n}u + nu^2}{1-u}\right) - 1 = \lambda\sqrt{n}u + O(u^2).$$

Then for $\rho_i = \pm 1$, $i = 1\colon n$ and any constant $\lambda > 0$,

$$\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1 + \theta_n, \quad |\theta_n| \le \widetilde{\gamma}_n(\lambda),$$

holds with probability at least $P(\lambda) = 1 - 2\exp(-\lambda^2/2)$.

Theorem 3.1 proves that the ubiquitous constant γ_n of backward error analysis can be replaced by a relaxed constant $\widetilde{\gamma}_n$ which only grows as $\lambda\sqrt{n}u$. This error bound holds with a certain probability $P(\lambda)$ which grows exponentially close to 1 when increasing λ; hence small values of λ (less than 10) suffice.

As a result, we obtain probabilistic analogues to many of the standard backward error bounds of linear algebra algorithms. In particular, we obtain backward error bounds for inner products, matrix multiplication, LU factorization, triangular solves, and the solution of linear systems all proportional to $\sqrt{n}u$ rather than nu (see Theorems 3.1–3.8 of **[1]**). Probabilistic backward error bounds for QR factorization can also be obtained with some more effort [99].

This new probabilistic analysis provides crucial stability guarantees for large scale and/or low precision computations. It also offers key new insights into the practical behavior of floating-point computations: indeed, by identifying in Model 3.1 conditions for the probabilistic bounds to hold, we also identify situations where these conditions are violated and therefore the worst-case bound is attained. These dangerous situations include what we have called in **[1]** "stagnation", a phenomenon where many small floating-point numbers that should be added to a larger one are absorbed and thus lost. This causes rounding errors to all be in the same direction and hence to lose their zero mean. Figure 3.1 illustrates that stagnation happens when summing a large number of nonnegative quantities: the partial sum eventually becomes so large that subsequent summands are absorbed. Stagnation starts happening for a number of summands of order $1/u$, which means that it happens sooner in lower precision.

In the recent years, the interest in probabilistic error analysis has reignited and has led to many other results bringing various extensions and refinements to the above results [169, 155, 156, 99, 67, 126, 264]. This is indeed a very active field and many open questions remain: see for example Research Problems 16.1 and 16.2.

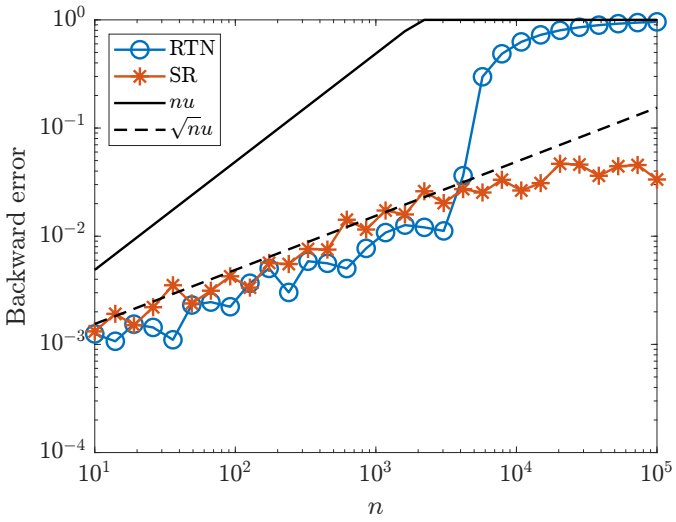


Figure 3.1: Backward error for computing the summation $s = \sum_{j=1}^{n} x_j$ in fp16 arithmetic, where the x_j are random independent variables sampled from the $[0, 1]$ uniform distribution, using two rounding modes: round-to-nearest (RTN) or stochastic rounding (SR).

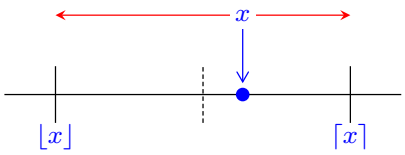


Figure 3.2: Stochastic rounding rounds x to one of the adjacent floating-point numbers $\lfloor x \rfloor$ and $\lceil x \rceil$ immediately below and above it, respectively. In this example, round-to-nearest would always round to $\lceil x \rceil$, whereas SR can round to either $\lfloor x \rfloor$ or $\lceil x \rceil$, but is more like to round to $\lceil x \rceil$.

3.2 Stochastic rounding

One important outcome of the above developments in probabilistic error analysis is that they provide a theoretical explanation for the success of stochastic rounding (SR). SR is a special rounding mode which rounds up or down not deterministically like other rounding modes, but rather with a certain well-chosen probability that decreases with the distance to each direction. Given $x \in \mathbb{R}$, the stochastically rounded number $\mathrm{SR}(x)$ is defined as

$$\mathrm{SR}(x) = \begin{cases} \lceil x \rceil & \text{with probability } p = \frac{x - \lfloor x \rfloor}{\lceil x \rceil - \lfloor x \rfloor} \\ \lfloor x \rfloor & \text{with probability } 1 - p \end{cases}, \tag{3.1}$$

where $\lfloor x \rfloor$ and $\lceil x \rceil$ are the adjacent floating-point numbers immediately below and above x, respectively. This is illustrated in Figure 3.2.

There has been a strong renewed interest in SR after it has been shown to increase the accuracy of some computations, including neural network training [147] (see Chapter 15). In our paper **[3]**, we build upon our probabilistic analysis described in the previous section to prove that SR actually enforces the rounding errors to satisfy Model 3.1: rounding errors produced by SR are mean independent, zero-mean random variables.

Theorem 3.2 (Lemma 5.2 in [3]). *The rounding errors generated by stochastic rounding satisfy Model 3.1. As a corollary, with stochastic rounding, Theorem 3.1 holds unconditionally.*

Theorem 3.2 shows that the probabilistic $\sqrt{n}u$ bound is guaranteed to hold unconditionally with SR, and this can therefore lead to an improvement of the accuracy by a factor $\sqrt{n}$ in the cases where the worst-case nu bound would be attained with deterministic rounding modes. Figure 3.1 illustrates this in the case of stagnation: unlike round-to-nearest, SR prevents small summands to be systematically absorbed by rounding up from time to time—with the exact probability required for the computed sum to be equal to the exact sum in expectation ([3], Theorem 6.2)!

This probabilistic error analysis therefore constitutes the first theoretical explanation of the success encountered by SR in low precision computations. For an in-depth review of the main properties of SR and its benefits in various applications, please also refer to our survey [4].

3.3 Probabilistic bounds for random data

The probabilistic bounds obtained in the previous sections only exploit a probabilistic model of the rounding errors: they do not make any assumption on the data. Further insights and reductions of the error bounds can be obtained by considering computations on random data. While real-life applications rarely involve random data, it is still important to understand the behavior of the error in this case, for several reasons.

- It is common practice to test, benchmark, and validate codes on randomly generated data. This is notably the case of the LINPACK benchmark used to make the TOP500 ranking.
- Some applications do involve random matrices; this is notably the case of randomized algorithms (see Section 6.2).
- Some applications involve nonrandom matrices that however resemble random ones; for example, the elements of the weight matrices in neural networks often follow a nearly Gaussian distribution (see Section 15.1).

In [2], we specialize Model 3.1 on rounding errors to the computation of a sum $s = \sum_{i=1}^{n} x_i$ with random independent summands x_i.

Model 3.2 (Models 2 and 3 in [2]). *Let x_j, $j = 1\colon n$ be random independent variables drawn from a distribution with mean μ_x satisfying $|x_j| \leq C_x$ for a given constant C_x, and let $\mu_{|x|}$ be the mean of the distribution of the $|x_j|$. Let $s = \sum_{j=1}^{n} x_j$ be computed by recursive summation, and assume that the rounding errors $\delta_1, \ldots, \delta_{n-1}$ generated in that order satisfy, for $k = 1\colon n-1$, $\mathbb{E}(\delta_k \mid \delta_1, \ldots, \delta_{k-1}, x_1, \ldots, x_n) = \mathbb{E}(\delta_k) = 0$.*

Theorem 3.3 (Corollary 2.10 in [2]). *Let $x \in \mathbb{R}^n$ let $s = \sum_{j=1}^{n} x_j$ be computed under Model 3.2. For any constant $\lambda > 0$ and for any n sufficiently large such that $\mu_{|x|}\sqrt{n} \geq 2\lambda C_x$, the computed $\widehat{s}$ satisfies the backward error bound*

$$\frac{|\widehat{s} - s|}{\sum_{j=1}^{n} |x_j|} \leq \frac{2}{\mu_{|x|}}(\lambda|\mu_x|\sqrt{n} + \lambda^2 C_x)u + O(u^2)$$

with probability at least $P(\lambda) = 1 - 2(n+1)\exp(-\lambda^2/2)$.

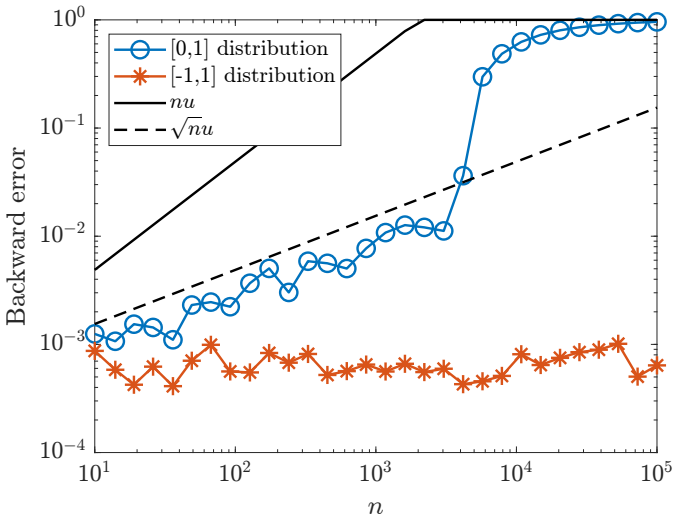


Figure 3.3: Backward error for computing the summation $s = \sum_{j=1}^{n} x_j$ in fp16 arithmetic, where the x_j are random independent variables sampled from the $[0, 1]$ or $[-1, 1]$ uniform distributions.

Theorem 3.3 shows that if the x_j are sampled for a centered (zero-mean) distribution, such as the commonly used standard normal and $[-1, 1]$ uniform distributions, then $\mu_x = 0$ and the $\sqrt{n}$ term in the bound vanishes. For such distributions, $\mu_{|x|}$ is a strictly positive constant and so the error bound is of order a constant independent of n times the unit roundoff u. This behavior of the backward error is illustrated in Figure 3.3. By taking into account the distribution and specifically the mean of the data, we can thus obtain sharper probabilistic bounds on the backward error.

These sharper bounds do not only apply to summation: in **[2]**, we derive similar bounds independent of n for inner products, matrix–vector products, and matrix–matrix products, whenever at least one of the involved vectors and/or matrices has its coefficients sampled from a zero-mean distribution (see Theorems 3.2, 3.3, and 3.4 of **[2]**). Unfortunately, the analysis does not readily extend to LU factorization and triangular solves, because of the divisions that they involve. Yet, in practice, the backward error for these computations is also observed to be independent of n, which suggests a similar mechanism is at play (see Research Problem 16.1).

3.4 Stochastic validation for inner products

Let $x, y \in \mathbb{R}^n$ and let $s = x^T y$ be their inner product. The computed $\widehat{s}$ satisfies the forward error bound

$$\widehat{s} = x^T y + \Delta s, \quad |\Delta s| \leq \gamma_n \kappa,$$

where

$$\kappa := \frac{|x|^T |y|}{|x^T y|}$$

is the condition number of the inner product.

In the previous sections, we have seen that the γ_n part of the bound is usually pessimistic but can be more sharply estimated via probabilistic analyses. The κ part of the bound is much harder to estimate reliably, because it requires the knowledge of the true inner product $x^T y$ itself. Yet, in some contexts it is important to be able to reliably estimate the accuracy of the

computed inner product—for example, in order to certify the correctness of a computation. This requires numerical validation tools; one way to develop such tools is to use stochastic arithmetic.

In stochastic arithmetic, all floating-point objects are represented by d distinct values (called representatives) and computations are performed on each of these d representatives with stochastic rounding. All intermediate quantities of the computation also have d representatives, and at the end of the computation, the accuracy of the final result can be estimated by comparing its d representatives: roughly, one can estimate that the digits that are identical for all representatives are correct, while the rest have been lost to numerical noise produced by rounding errors. This approach is for example implemented in the CADNA library [173].

Stochastic arithmetic is however expensive, not so much because each operation is repeated d times, but more importantly because each elementary operation must be performed separately, in order to apply stochastic rounding. Thus, in the computation of an inner product, stochastic rounding must be applied after each addition and multiplication. This is a major performance hurdle because it prevents the use of optimized libraries, such as the BLAS, which do not support stochastic rounding modes.

In **[38]**, we propose a new method to overcome this hurdle. The main idea is to avoid introducing randomness in the intermediate steps of the computation: the method only adds random perturbations to the input x (and/or y). It therefore presents the significant advantage of not requiring intrusive modifications of the intermediate computations of the inner product, and can thus rely on optimized implementations, such as the BLAS. Specifically, we introduce randomness in the input by randomly perturbing each representative of x as follows:

$$x^{(1)} = x \circ (1 + \delta\xi^{(1)}), \tag{3.2}$$

$$\cdots$$

$$x^{(d)} = x \circ (1 + \delta\xi^{(d)}), \tag{3.3}$$

where $\circ$ denotes the Hadamard (componentwise) product, $\delta > 0$, and $\xi^{(1)}, \ldots, \xi^{(d)} \in \mathbb{R}^n$ are standard normal random vectors, that is, their elements $\xi_k^{(i)}$ are drawn from the standard normal distribution $\mathcal{N}(0,1)$. Then, we compute the d inner products $s^{(i)} = (x^{(i)})^T y$ with classical deterministic arithmetic and we define their average representative

$$\bar{s} = \frac{1}{d}\sum_{i=1}^{d} s^{(i)}. \tag{3.4}$$

The following result proves that the accuracy of $\bar{s}$ can be reliably estimated by (3.5) below.

Theorem 3.4 (Theorem 4.3 in **[38]**). *Let $\bar{s}$ in (3.4) be computed in deterministic floating-point arithmetic with unit roundoff u. Let*

$$\varepsilon = \sqrt{\frac{\sum_{i=1}^{d}(\bar{s} - s^{(i)})^2}{d(d-1)}}. \tag{3.5}$$

Then for any probability $\beta > 0$ the computed $\widehat{\bar{s}}$ satisfies

$$\mathbb{P}(|s - \widehat{\bar{s}}| \leq 2\tau_\beta \varepsilon) \geq \beta\left(1 - c\left(\frac{nu}{\delta}\right)^{(d-1)/2}\right) \tag{3.6}$$

where τ_β is the value of Student's distribution with $d-1$ degrees of freedom and probability level β, and c is a modest constant.

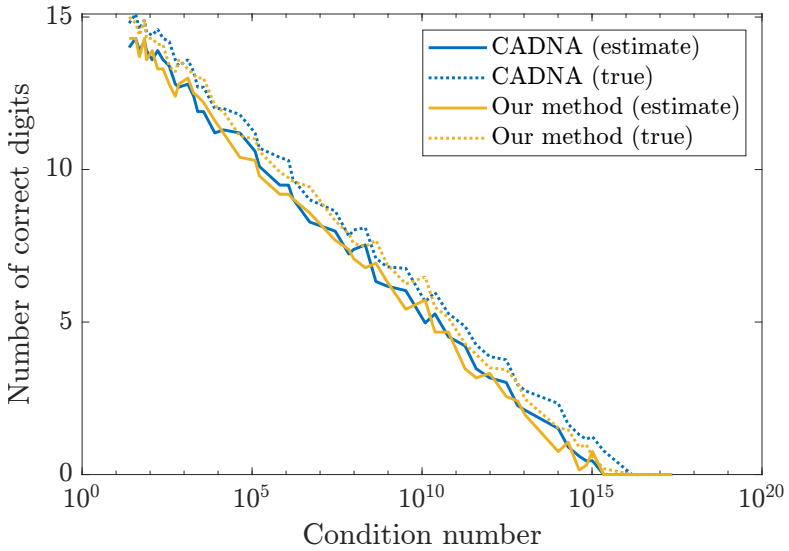


Figure 3.4: Estimated and true accuracy of inner products computed by CADNA and by our method with $\delta = \sqrt{n}u = 10u$, depending on the condition number.

The bound (3.6) shows that ε is a reliable estimate as long as the size of the random perturbations δ is sufficiently larger than nu. Moreover, taking into account the statiscal arguments developed in the previous sections, taking $\delta = \sqrt{n}u$ is often sufficient. Figure 3.4 illustrates the reliability of our method for vectors of length $n = 100$ and varying condition number. We compare the standard CADNA method, which implements stochastic arithmetic, and our method with deterministic arithmetic and a random perturbation of size $\delta = 10u$. The figure confirms that our method provides a conservative estimate; moreover, the introduced perturbation is small enough that its accuracy remains comparable to that of CADNA.

Chapter 4

Summation and matrix multiplication

Computing sums of floating-point numbers is one of the fundamental underlying kernels of matrix-based computations. Consider for example the inner product of two vectors $x, y \in \mathbb{R}^n$

$$s = x^T y = \sum_{i=1}^{n} x_i y_i \tag{4.1}$$

(throughout this chapter we consider inner products, but the results apply trivially to plain summation by setting y to the vector of all ones). Recursive summation evaluates the sum in (4.1) in the natural order, and yields a computed $\widehat{s}$ that satisfies the backward error bound [162]

$$\widehat{s} = (x + \Delta x)^T y, \quad |\Delta x| \leq \gamma_n |x|. \tag{4.2}$$

This implies the absolute forward error bound

$$\widehat{s} = s + \Delta s, \quad |\Delta s| \leq \gamma_n |x|^T |y| \tag{4.3}$$

and the relative forward error bound

$$\frac{|\widehat{s} - s|}{|s|} \leq \gamma_n \kappa \tag{4.4}$$

where

$$\kappa = \frac{|x|^T |y|}{|x^T y|} \tag{4.5}$$

is the condition number of the inner product.

As (4.4) shows, there are two main challenges in accurately computing inner products: dealing with a large number of summands n (the challenge of rounding error accumulation), and dealing with a large condition number κ (the challenge of cancellation: summands of opposite sign cancel each other, making $|x|^T |y|$ much larger than $|x^T y|$).

This chapter describes summation algorithms that tackle either one of these challenges. In Section 4.1, we analyze the numerical properties of specialized mixed precision hardware such as GPU tensor cores that can accumulate 16-bit numbers in 32-bit arithmetic, and show how this improves the error bounds. In Section 4.2, we describe various tree-based summation algorithms that achieve reduced error bounds, including a mixed precision blocked summation algorithm called FABsum **[19]**. Finally, in Section 4.3 we discuss distillation algorithms for evaluating highly ill-conditioned sums.

As discussed in the previous chapter, the worst-case error bounds can be pessimistic and on average we may replace them by their square root. However, bounds proportional to $\sqrt{n}$ can still be unsatisfactory for extreme scale sums and/or low precision arithmetic, so that we cannot only rely on probabilistic effects to guarantee the accuracy of the result. Fortunately, the general probabilistic backward error analysis developed in Chapter 3 applies to all algorithms described in the present chapter, so that we can combine the probabilistic effect with the reduction of the worst-case bound.

The contributions described in this chapter are based on the following papers: **[5]**, **[6]** for Section 4.1; **[19]** for Section 4.2; and **[20]** for Section 4.3.

4.1 Mixed precision matrix multiply–accumulate

A wide range of modern hardware provides specialized mixed precision matrix multiplication units that take as input low precision matrices and accumulate the result using higher precision arithmetic. This is in particular the case of NVIDIA GPUs, which are equipped with so-called tensor cores that can compute the product $C = AB$ where A and B are stored in half precision (fp16 or bfloat16) and the intermediate computations are accumulated in C using single precision (fp32) arithmetic. There is a growing range of hardware with similar mixed precision features, such as Google's TPUs, AMD's MatrixCores, and ARM's v8-A CPUs. Due to its specialized and intrinsically mixed precision nature, the numerical behavior of such hardware is not covered by standard rounding error analysis, and can indeed depart significantly from the usual behavior of standard floating-point units following IEEE arithmetic. To analyze their common properties and better understand their numerical behavior, we define a general model that encompasses all these different units.

Model 4.1. *Given matrices $A \in \mathbb{R}^{b_1 \times b}$ and $B \in \mathbb{R}^{b \times b_2}$ stored in precision u_{low}, we compute $C = AB$ in precision u_{high}.*

Model 4.1 was coined a "block fused multiply–add" (block FMA) in **[5]** because, even though it does not actually implement an FMA, its numerical behavior can resemble one when $u_{\text{high}} \ll u_{\text{low}}$: in this case, intermediate rounding errors of size u_{high} are negligible in front of the conversion errors of size u_{low} incurred when casting A and B to low precision.

It is important to note that, while Model 4.1 captures the essential properties of the previously mentioned hardware examples, some hardware might not exactly follow this model, or might have other specific features (such as a particular rounding mode or order of operations) that are not included in the model. In fact, in many cases, the behavior of such hardware has not been precisely documented or specified by the vendors, and it is often necessary to turn to experiments to better understand their properties [128].

Note that Model 4.1 assumes fixed matrix sizes $b_1 \times b$ and $b \times b_2$, in conformity with the examples mentioned above (for example, $b_1 = b = b_2 = 4$ for NVIDIA tensor cores). However, such a kernel can readily be used to compute matrix products $C = AB$ of arbitrary dimensions by partitioning A and B in blocks of size $b_1 \times b$ and $b \times b_2$, respectively, and computing a blocked matrix product; see Algorithm 3.1 of **[5]** for a detailed description.

Theorem 4.1 (Theorem. 3.2 of **[5]**). *Let the product $C = AB$ of $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times t}$ be evaluated using a mixed precision matrix multiplication unit satisfying Model 4.1. The computed $\widehat{C}$ satisfies*

$$|C - \widehat{C}| \leq \left(2u_{\text{low}} + u_{\text{low}}^2 + \gamma_n^{\text{high}}(1 + u_{\text{low}})^2\right)|A||B|, \tag{4.6}$$

where $\gamma_n^{\text{high}} = nu_{\text{high}}/(1 - nu_{\text{high}})$.

Theorem 4.1 shows that the mixed precision nature of Model 4.1 can be harnessed to significantly improve the error bound of matrix multiplication: the dependence on n due to rounding error accumulation only affects the $O(u_{\text{high}})$ part of the bound, the $O(u_{\text{low}})$ part being independent of n. This is a valuable property that can significantly improve the accuracy of the result; we will discuss its impact on LU factorization in Section 10.1.2.

Theorem 4.1, like most traditional error analyses, assumes the absence of overflow and underflow. However, some of the low precision input formats required by such hardware offer a much narrower range of representable values. This issue becomes more pressing on recent GPU models like NVIDIA Hopper, which support quarter precision (8-bit) arithmetic; 6-bit and 4-bit formats have even been announced on the forthcoming NVIDIA Blackwell. Therefore, it is clear that range issues can no longer be ignored on this type of hardware.

This motivates two new developments:

- the incorporation of scaling techniques within standard linear algebra computations in order to prevent overflow and minimize underflow;
- and new error analyses taking into account the effect of underflow, because even with scaling the range of these new formats is so narrow that some amount of underflow is inevitable.

To account for the range limitations, let us extend Model 4.1 as follows.

Model 4.2. *Let $A \in \mathbb{R}^{b_1 \times b}$ and $B \in \mathbb{R}^{b \times b_2}$ be stored in a low precision input format with unit roundoff u_{low} and range $[f_{\text{low}}^{\min}, f_{\text{low}}^{\max}]$. We compute $C = AB$ in a high precision accumulation format with unit roundoff u_{high} and range $[F_{\text{high}}^{\min}, F_{\text{high}}^{\max}]$.*

Since the coefficients of A and B may not belong to the range of the low precision input format, we must scale them. We compute

$$C = \Lambda^{-1}\Big(\mathrm{fl}_{\text{low}}(\Lambda A)\,\mathrm{fl}_{\text{low}}(BM)\Big)M^{-1} \tag{4.7}$$

where Λ and M are diagonal matrices and where $\mathrm{fl}_{\text{low}}(\cdot)$ denotes the rounding operator to the low precision format. Let θ be the maximum value that we wish to allow in ΛA and BM. There exists an optimal value for θ: as it increases, underflow is minimized, so we must take its largest possible value which prevents overflow from occurring both in the input and in the intermediate computations. Preventing overflow in the input requires $\theta \le f_{\text{low}}^{\max}$, whereas preventing it in the accumulation requires $\theta \le \sqrt{F_{\text{high}}^{\max}/n}$. We therefore should set

$$\theta = \min(f_{\text{low}}^{\max}, \sqrt{F_{\text{high}}^{\max}/n}). \tag{4.8}$$

The following result provides an analogue to Theorem 4.1 which takes into account the range limitations.

Theorem 4.2 (Theorem 3.1 of **[6]**). *Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ and let Λ and M be diagonal scaling matrices such that the elements of ΛA and BM are all bounded by θ in (4.8). Let $C = AB$ be evaluated as in (4.7) using a mixed precision unit satisfying Model 4.2. Then the computed $\widehat{C}$ satisfies*

$$\|\widehat{C} - AB\|_\infty \lesssim \Big(2u_{\text{low}} + nu_{\text{high}} + 2n^2\theta^{-1}f_{\text{low}}^{\min} + 4n^2\theta^{-2}F_{\text{high}}^{\min}\Big)\|A\|_\infty\|B\|_\infty. \tag{4.9}$$

In presence of underflow, we can only obtain a normwise bound. This bound (4.9) should be compared to the normwise bound

$$\|\widehat{C} - AB\|_\infty \lesssim (2u_{\text{low}} + nu_{\text{high}})\|A\|_\infty\|B\|_\infty \tag{4.10}$$

that follows from (4.6) in Theorem 4.1. Bound (4.9) thus recovers the term $2u_{\text{low}}+nu_{\text{high}}$ coming from the rounding errors. In addition, it also accounts for the errors caused by underflows with the terms $2n^2\theta^{-1}f_{\text{low}}^{\min}$ (input underflows) and $4n^2\theta^{-2}F_{\text{high}}^{\min}$ (accumulation underflows).

Can these underflows become a cause of concern? There cannot be a definitive answer in general, because it depends on the properties of the input and accumulation formats, as well as the inner dimension n of the matrices. However, note that if $F_{\text{high}}^{\max}$ is sufficiently large so that $f_{\text{low}}^{\max} \leq \sqrt{F_{\text{high}}^{\max}/n}$, then $\theta = f_{\text{low}}^{\max}$ and thus the term $\omega = \theta^{-1}f_{\text{low}}^{\min} = f_{\text{low}}^{\min}/f_{\text{low}}^{\max}$ corresponds to the inverse width of the range of the input format. For all practical formats of interest, ω is smaller than the corresponding unit roundoff u_{low} and hence we can expect the rounding errors to remain the dominant source of errors. Moreover, Theorem 4.2 does not assume any support for denormalized numbers. If this assumption is added, gradual underflow significantly improves the bound (4.9), in which the terms $f_{\text{low}}^{\min}$ and $F_{\text{high}}^{\min}$ are replaced by $2u_{\text{low}}f_{\text{low}}^{\min}$ and $2u_{\text{high}}F_{\text{high}}^{\min}$, respectively.

Overall, in practice, for most cases of interest the underflows should thus remain dominated by the rounding errors, and therefore the range limitations should not significantly affect the accuracy of the computation—provided a suitable scaling is used, naturally.

4.2 Blocked summation

The constant n in the bound (4.2) comes from the fact that the summands x_i and y_i are involved in at most n operations: 1 multiplication and at most $n-1$ additions. This means that the computed $\widehat{s}$ satisfies

$$\widehat{s} = \sum_{i=1}^{n} x_i y_i \prod_{k=1}^{k_i}(1+\delta_{i,k})$$

with $k_i \leq n$ and $|\delta_{i,k}| \leq u$, and thus (4.2) follows from Theorem 2.2.

Therefore, a popular technique for reducing the constants in the error bounds for summation is to reduce the maximum number of operations that any summand is involved with. For example, blocked summation consists in computing partial sums of blocks of b summands $s_j = \sum_{i=(j-1)b+1}^{jb} x_i y_i$ before summing together the n/b partial sums $s = \sum_{j=1}^{n/b} s_j$. In this case, any summand $x_i y_i$ is involved in at most $b+n/b-1$ operations: 1 multiplication, $b-1$ additions in the computation of the partial sum, and $n/b-1$ additions in the computation of the final sum. We thus obtain a reduced backward error bound

$$\widehat{s} = (x+\Delta x)^T y, \quad |\Delta x| \leq \gamma_{b+n/b-1}|x| \tag{4.11}$$

where the constant attains it minimal value $\gamma_{2\sqrt{n}-1}$ for the optimal block size $b = \sqrt{n}$.

While potentially useful, one level of blocking is therefore not sufficient to handle very large sums: we would like to further reduce the dependence of the error on n. This is possible by using more general summation trees. A summation tree is a binary tree that defines the order in which the summands are added together: the summands correspond to the leaf nodes, and all the other nodes are defined as the sum of their two children. The root node therefore corresponds to the final sum. With this formalism, recursive summation corresponds to a comb tree of height $n-1$, whereas blocked summation corresponds to a block comb tree of height $n/b-1$ where

each block node is a comb tree of height $b-1$. A generalization with $t>1$ levels of blocking is described by Castaldo et al. [93]. The smallest possible bound is obtained by minimizing the height of the summation tree, that is, by using a balanced tree of height of $\lceil \log_2 n \rceil$; this is called pairwise summation [162, Chap. 4]. Unfortunately, pairwise summation is inefficient on modern hardware as it prevents the use of vectorization and other blocking techniques. For these reasons, in practice, linear algebra algorithms use blocked summation with usually one or two levels of blocking.

Algorithm 4.1 Fast and accurate blocked summation (FABsum).

Input: $x, y \in \mathbb{R}^n$, a block size b, and two summation algorithms `FastSum` and `AccurateSum`.
Output: $s = x^T y$.

1: **for** $j = 1 : n/b$ **do**
2: Compute $s_j = \sum_{i=(j-1)b+1}^{jb} x_i y_i$ with `FastSum`.
3: **end for**
4: Compute $s = \sum_{j=1}^{n/b} s_j$ with `AccurateSum`.

To overcome the dependence on n of the error while preserving efficiency, we propose in **[19]** the FABsum (Fast and Accurate Blocked summation) algorithm, outlined in Algorithm 4.1. While FABsum uses only one level of blocking, it uses two different summation algorithms: a fast algorithm `FastSum` to compute the partial sums of size b, and an accurate one `AccurateSum` to sum the n/b partial sums. Computationally speaking, this is efficient because the accurate algorithm performs only n/b flops out of n (for example, for $b = 100$, only 1% of the flops). Moreover, we can bound the backward error for FABsum as a function of the backward errors of `FastSum` and `AccurateSum`. Indeed, assume that we have the backward error bound

$$\widehat{s} = (x+\Delta x)^T y, \quad \begin{cases} |\Delta x| \le \varepsilon_f(n)|x| & \text{with } \texttt{FastSum} \\ |\Delta x| \le \varepsilon_a(n)|x| & \text{with } \texttt{AccurateSum} \end{cases}. \tag{4.12}$$

Then we have the following result for FABsum.

Theorem 4.3 (Theorem. 3.1 of **[19]**). *Let $s = x^T y$ be computed by Algorithm 4.1. Assuming (4.12), the computed $\widehat{s}$ satisfies*

$$\widehat{s} = (x+\Delta x)^T y, \quad |\Delta x| \le \Big(\varepsilon_f(b) + \varepsilon_a(n/b) + \varepsilon_f(b)\varepsilon_a(n/b)\Big)|x|. \tag{4.13}$$

Crucially, in (4.13) the `FastSum` part of the bound $\varepsilon_f(b)$ does not depend on n, only on b. Thus, we may absorb the dependence on n by using an accurate algorithm for `AccurateSum` such as using extra precision or compensated summation. In particular, if we use recursive summation for both algorithms, in precision u for `FastSum` and in precision u^2 for `AccurateSum`, we obtain a mixed precision version of FABsum that achieves a backward error bound $bu+O(u^2)$ independent of n to first order.

We refer to **[19]** for numerical examples illustrating how FABsum can improve the accuracy of matrix-based computations; see also Section 5.2 for a situation where the use of FABsum is important.

4.3 Distillation of ill-conditioned sums

While the previous sections have dealt with the problem of reducing rounding error accumulation, we now turn to the problem of handling ill-conditioned sums. Several methods have

been proposed to tackle this problem, the most popular of which we can categorize in two broad classes.

- The first class relies on the finite number of exponents in typical (IEEE) floating-point arithmetic and on the fact that numbers with exponents not too far apart can be added exactly using extended precision accumulators. This is for example the case of Kulisch's accumulator [178], which sums all numbers in a very long accumulator, or of the Demmel–Hida algorithm [108], which sums together numbers of comparable exponent using higher precision arithmetic, such as quadruple precision. This strategy is also used in HybridSum [262] and OnlineExactSum [261].

- The second class exploits the fact that the error incurred by floating-point addition is itself a floating-point number. This error can be computed exactly via error-free transformations such as Fast2Sum. This class includes in particular the AccSum method [228], which computes a faithful rounding of the sum irrespective of its conditioning, and the PrecSum method [229], which computes the sum as if using K-fold precision (for a given K). Those algorithms have been improved respectively as FastAccSum and FastPrecSum in [227].

The methods from the first class have a double weakness: they require access to the exponent of the floating-point numbers, which can be expensive, and they require the use of extended precision accumulators. In contrast, the methods from the second class only require standard arithmetic operations on the summands. However, they also require a much larger number of floating-point operations, and their cost strongly depends on the conditioning. Moreover, they are much less parallel than the methods from the first class. In any case, both classes of methods can be quite expensive.

Many of the summation methods able to handle ill-conditioned sums rely on the process of *distillation*. Distillation consists in iteratively transforming the original, ill-conditioned sum into an equivalent but well-conditioned sum that can then be evaluated accurately. This is especially the case of the second class of methods mentioned above, although the first class can also be used to design distillation methods, see for example [108, sect. 5] for the Demmel–Hida method.

In **[20]**, we propose a method to transform the original sum into another, equivalent sum, which is not necessarily better conditioned, but which is far smaller. The smaller sum can then be distilled inexpensively by any of the distillation methods mentioned above. We call the first transformation of the sum into a smaller equivalent one the process of *condensation*.

The condensation step is based on the following key observation.

Theorem 4.4 (Corollary 2.3 of **[20]**). *If x, y are base-two floating-point numbers with the same sign, exponent, and least significant bit, then barring overflow their addition is exact.*

Theorem 4.4 gives a sufficient condition for the addition of two floating-point numbers to be exact, even *in the working precision*. We harness this observation to develop a fast distillation algorithm, called Condense & Distill, which consists of two steps. The first step is to efficiently condense the sum by summing pairs of summands with the same sign, exponent, and least significant bit, until no such pairs remain. The number of remaining summands is necessarily bounded, and the second step is to distill them with classical distillation algorithms.

Compared with the Demmel–Hida method, Condense & Distill is also based on adding numbers with the same exponent, but its key advantage is that it does not require any extra precision. The whole condensation process can indeed be performed entirely in the working precision. This is achieved at the cost of accessing the least significant bit of the summands, which is a negligible overhead compared with the cost of accessing their exponent. Compared with the second class of summation methods (AccSum, etc.), Condense & Distill shares the main strengths of the first

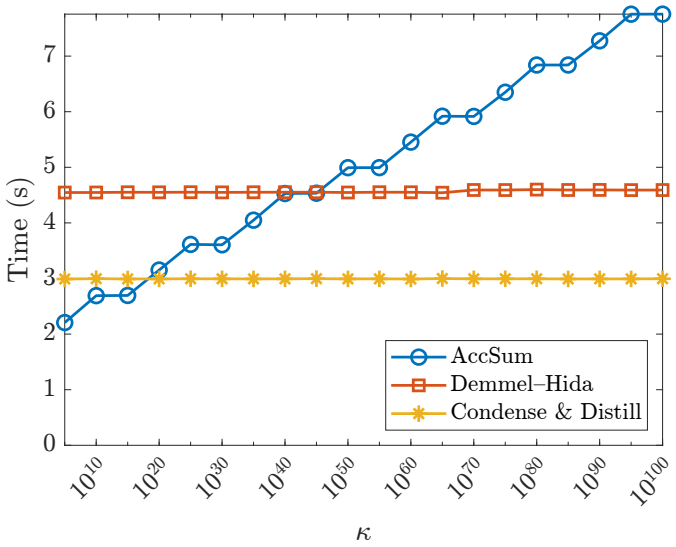


Figure 4.1: Comparison between the Demmel–Hida, AccSum, and Condense & Distill algorithms to distill a sum with $n = 10^8$ summands, as a function of its condition number κ. All algorithms are run sequentially (1 thread).

class: its performance is completely independent of the conditioning of the sum, and it exhibits nearly perfect parallel scaling.

We illustrate the attractive properties of Condense & Distill in Figure 4.1, which shows that this new method is about 35% faster than the Demmel–Hida method, and that it outperforms the AccSum method when the sum is sufficiently ill-conditioned. See **[20]** for more performance results, including parallel scaling experiments.

Chapter 5

Multiword arithmetic

This chapter deals with the fundamental concept of multiword arithmetic, which allows for emulating high precision arithmetic while relying on low precision computations.

After reviewing the basic principles of multiword arithmetic in Section 5.1, we discuss two situations in which it presents a high potential for improving matrix multiplication.

Section 5.2 describes the first situation, which is related to the use of specialized mixed precision hardware such as GPU tensor core units (see Section 4.1). With such hardware, the unusually large speedups between low and high precision computations, combined with the ability to accumulate low precision numbers in high precision arithmetic, makes multiword arithmetic based on low precision words as accurate but much faster than standard high precision arithmetic.

Section 5.3 describes the second situation, which is related to computing matrix multiplication over prime finite fields. In this case, all the matrix coefficients are integers, which means the problematic becomes not one of precision (computations with integers being exact), but one of range (ensuring that all integers arising from the computation do not overflow, that is, remain smaller than the largest representable integer in the working arithmetic). We show how to leverage multiword arithmetic in this context to handle large integers efficiently.

The contributions described in this chapter are based on the following papers: **[7]**, **[6]** for Section 5.2 and **[45]** for Section 5.3.

5.1 Basics

Given a working precision arithmetic with t bits of significand, multiword arithmetic represents a real number $x \in \mathbb{R}$ as the unevaluated sum of non-overlapping t-bit numbers (called "words"). For example, two-word arithmetic represents

$$x \approx x_1 + x_2, \tag{5.1}$$

where x_1 and x_2 each have t bits of significand. If x_1 is obtained by rounding x to the nearest floating-point number with t bits of significand, then barring overflow (5.1) approximates x as a number with at least $2t + 1$ bits of significand.

When $t = 53$ (double precision as the working precision arithmetic), (5.1) corresponds to double-double arithmetic. Double-double arithmetic is historically the most common and widely used form of multiword arithmetic. It delivers an accuracy corresponding to 107 bits of significand, that is, $2^{-107} \approx 6 \times 10^{-33}$, and is thus slightly less accurate than quadruple precision arithmetic (fp128, which uses 113 bits of significand, delivering an accuracy of $2^{-113} \approx 1 \times 10^{-34}$). However, quadruple precision is not supported natively by most processors, and so double-double

arithmetic is potentially faster because the underlying operations all use the natively supported double precision arithmetic.

That being said, double-double arithmetic is significantly more expensive than double precision arithmetic. This is not only because the number of flops increases with the number of words, but more importantly because preserving the accuracy of the computation requires special handling to avoid rounding errors of size the working precision. This is achieved via various techniques based on error-free transformations, of which we omit a detailed description; please refer instead to [205, Chap. 14]. The important point is that these techniques further increase the number of flops, making double-double arithmetic at least an order of magnitude more expensive than double precision arithmetic on most computer architectures.

For this reason, traditional multiword arithmetic has mainly been used to enhance the accuracy when the highest available precision was not sufficient. The next two sections describe two situations in which intermediate computations do not lead to rounding errors of size the working precision. In these situations, multiword arithmetic can then be implemented with standard floating-point operations and is thus particularly beneficial, even with lower precision arithmetics as the working precision.

5.2 Multiword matrix multiplication with mixed precision hardware

Multiword arithmetic allows for achieving both high accuracy and high performance when combined with specialized mixed precision hardware available on modern accelerators, such as GPU tensor cores.

Indeed, as discussed and analyzed in Section 4.1, such units can multiply low precision (16-bit) matrices while accumulating all intermediate rounding errors in higher precision (32-bit) arithmetic. Multiword arithmetic can be used to remove the low precision conversion error term by representing the matrices on several words. In particular, consider the two-word decompositions $A \approx A_0 + A_1$ and $B \approx B_0 + B_1$, where A_0, A_1, B_0, and B_1 are all stored in fp16. Then $C = AB$ can be approximated as four products $A_0B_0 + A_1B_0 + A_0B_1 + A_1B_1$, each of which can be efficiently computed using the mixed precision hardware with fp32 accumulation. This idea has first been proposed by Markidis et al. [194] to exploit the speed of the NVIDIA GPU tensor cores while enhancing their accuracy. Henry et al. [158] investigate a similar approach but using bfloat16 instead of fp16. Since bfloat16 only uses 8 bits for the significant, three words are necessary to emulate full fp32 accuracy. Mukunoki et al. [204] propose an approach to emulate fp64 accuracy using NVIDIA tensor cores. Their approach is based on the Ozaki scheme [217]: the matrices are decomposed in sufficiently many words so that multiplying any two words can be done exactly. This approach is closer in spirit to the traditional use of multiword arithmetic since it requires a large number of words. Ootomo et al. [210] show how to extend this approach to use low precision integer arithmetic, and Uchino et al. [242] propose some improvements to their approach. Most recently, Ozaki et al. [218] have proposed a second scheme to emulate high precision floating-point arithmetic with low precision integer arithmetic based on a very different approach. This "Ozaki scheme II" computes the product modulo several coprime numbers that fit in the low precision, and recovers the high precision product using the Chinese remainder theorem; this second scheme is shown to require fewer matrix products than the first Ozaki scheme.

Multiword arithmetic can also be used in more complex computations than just matrix multiplication: for example, Sorna et al. [237] use it in the fast Fourier transform, and Ootomo and Yokota [212] use it in the randomized SVD (see Section 6.4 for more details).

These approaches can be made general by considering an arbitrary number of words w and

an abstract mixed precision hardware satisfying Model 4.1. For $i, j = 0\colon w-1$, we define

$$A_i = \mathrm{fl}_{\mathrm{low}}\left(\Big(A - \sum_{k=0}^{i-1} u_{\mathrm{low}}^k A_k\Big)/u_{\mathrm{low}}\right), \qquad B_j = \mathrm{fl}_{\mathrm{low}}\left(\Big(B - \sum_{k=0}^{j-1} u_{\mathrm{low}}^k B_k\Big)/u_{\mathrm{low}}\right), \tag{5.2}$$

where $\mathrm{fl}_{\mathrm{low}}(\cdot)$ is the operator that rounds to precision u_{low}. This yields the decompositions

$$A + \Delta A = \sum_{i=0}^{w-1} u_{\mathrm{low}}^i A_i, \quad |\Delta A| \le u_{\mathrm{low}}^w |A|,$$
$$B + \Delta B = \sum_{j=0}^{w-1} u_{\mathrm{low}}^j B_j, \quad |\Delta B| \le u_{\mathrm{low}}^w |B|,$$

with an error of order u_{low}^w. Since all the A_i and B_j are stored in precision u_{low}, the product $C = AB$ can be computed with a mixed precision hardware satisfying Model 4.1 as

$$C = \sum_{i=0}^{w-1} \sum_{j=0}^{w-1} u_{\mathrm{low}}^{i+j} A_i B_j$$

with all intermediate computations performed in precision u_{high}. Importantly, the term u_{low}^{i+j} shows that not all w^2 products $A_i B_j$ need be computed: we can ignore any product $A_i B_j$ with $i + j \ge w$ while introducing an error of order at most $O(u_{\mathrm{low}}^w)$. This reduces the number of products from w^2 to $w(w+1)/2$.

Algorithm 5.1 Multiword matrix multiplication with mixed precision hardware.

Input: Two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ and a number of words w.
Output: $C = AB$ computed with w-word arithmetic.

for $i = 0$ **to** $w-1$ **do**
  $A_i = \mathrm{fl}_{\mathrm{low}}((A - \sum_{k=0}^{i-1} u_{\mathrm{low}}^k A_k)/u_{\mathrm{low}})$
  $B_i = \mathrm{fl}_{\mathrm{low}}((B - \sum_{k=0}^{i-1} u_{\mathrm{low}}^k B_k)/u_{\mathrm{low}})$
end for
$C = 0$
for $i = 0$ **to** $w-1$ **do**
  for $j = 0$ **to** $w-1$ **do**
    if $i + j < w$ **then**
      $C = C + u_{\mathrm{low}}^{i+j} A_i B_j$ with a mixed precision hardware satisfying Model 4.1.
    end if
  end for
end for

Algorithm 5.1 summarizes how to compute $C = AB$ in w-word arithmetic using mixed precision hardware. The following result shows that the accuracy of the computed $\widehat{C}$ is of order $u_{\mathrm{low}}^w + u_{\mathrm{high}}$.

Theorem 5.1 (Theorem 2.1 of [7]). *Let $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times q}$ and let $C = AB$ be computed by Algorithm 5.1. The computed $\widehat{C}$ satisfies*

$$|\widehat{C} - C| \le \Big((w+1)u_{\mathrm{low}}^w + \gamma_{n+w^2-1}^{\mathrm{high}}\Big)|A||B| + O(u_{\mathrm{high}} u_{\mathrm{low}} + u_{\mathrm{low}}^{w+1}).$$

In particular, on NVIDIA GPU tensor cores where u_{low} and u_{high} correspond to fp16 and fp32 arithmetics, Theorem 5.1 shows that $w = 2$ words suffice to recover an accuracy of order $u_{\text{low}}^2 \approx u_{\text{high}} \approx 10^{-7}$ similar to standard fp32 arithmetic. This "double-fp16" approach requires three products, and hence three times as many flops, but thanks to the much higher speed of the tensor cores units, it can significantly outperform standard fp32 arithmetic. We report performance results on A100 GPUs in **[7]** where double-fp16 arithmetic is up to 7× faster than fp32 arithmetic, while achieving a comparable accuracy.

However, we also identify in **[7]** a situation in which double-fp16 arithmetic can fail to deliver the expected accuracy due to a numerical issue specific to NVIDIA GPU tensor cores. This hardware uses the round-to-zero mode [128], which tends to produce rounding errors whose mean strongly deviates from zero. This exacerbates the effect of rounding error accumulation, whereas standard fp32 arithmetic uses the round-to-nearest mode that benefits from an attenuated accumulation due to the probabilistic effects discussed in Chapter 3. Thus, while the double-fp16 and fp32 approaches share the same worst-case error *bound*, they do not achieve the same *error*.

There is a clear remedy to this issue: reduce the worst-case bound, so that the fact that it is attained with round-to-zero is no longer an issue. To do so, we can replace the underlying summation algorithm used by the matrix products by the more accurate tree-based summation algorithms discussed in Chapter 4. In particular, we show in **[7]** that the use of FABsum (Algorithm 4.1) allows the double-fp16 approach to recover a 32-bit accuracy while still outperforming standard fp32 arithmetic. A similar idea is also proposed by Ootomo and Yokota [211].

This result connects together three pieces of work from seemingly unrelated topics: probabilistic error analysis (Chapter 3), summation algorithms (Chapter 4), and multiword arithmetic (this chapter). This illustrates how the emergence of high performance low precision arithmetics can bring about the convergence of various fields such as high performance computing, computer arithmetic, and rounding error analysis.

5.3 Multiword matrix multiplication over prime finite fields

Many computer algebra applications require linear algebra computations over prime finite fields, that is, over the modular integers $\mathbb{Z}/p\mathbb{Z}$, where p is a prime number. Thus the computation of the modular matrix product

$$C = AB \bmod p, \tag{5.3}$$

where the coefficients of the matrices are all integers, is a fundamental basic block of computer algebra. In many cases, it is desirable to accommodate for the largest possible prime p.

In floating-point arithmetic, additions and multiplications with integers do not incur rounding errors, and hence matrix products are exact provided that all intermediate quantities remain representable as floating-point numbers, that is, provided they do not exceed 2^t, the maximum value for which all integers are exactly representable as floating-point numbers with t bits of significand. The problematic thus becomes not one of precision, but one of range.

In order to ensure that the integers arising in the computation of the product remain representable, we use modular arithmetic: we apply modular reductions with respect to p to keep the integers bounded throughout the computation. A naive implementation would be to simply perform a reduction after each floating-point operation: given $A \in \mathbb{Z}^{m\times n}$ and $B \in \mathbb{Z}^{n\times q}$, $C = AB \bmod p$ can be computed as

$$C \leftarrow C + (a_k b_k^T \bmod p) \bmod p, \quad k = 1\colon n, \tag{5.4}$$

where $a_k \in \mathbb{Z}^m$ is the kth column of A and $b_k^T \in \mathbb{Z}^q$ is the kth row of B. This approach is however extremely inefficient since it requires as many reductions as floating-point operations.

The number of reductions can be reduced by computing instead

$$C \leftarrow C + (A_K B_K \bmod p) \bmod p, \quad K = (k-1)\lambda + 1 \colon \min(k\lambda, n), \quad k = 1 \colon \lceil n/\lambda \rceil, \tag{5.5}$$

where $A_K \in \mathbb{Z}^{m \times \lambda}$ and $B_K \in \mathbb{Z}^{\lambda \times q}$ are block-columns of A and block-rows of B, respectively, and where λ is a block size that controls how often the reductions are performed. To maximize the efficiency, we want to take λ as large as possible while ensuring that the intermediate computations do not overflow. Assuming that the coefficients of A and B are already reduced modulo p, then the coefficients of $A_K B_K$ are bounded by $\lambda(p-1)^2$ and so this leads to $\lambda = \lfloor 2^t/(p-1)^2 \rfloor$. This approach is for example implemented in the FFLAS-FFPACK library [121].

While this approach is efficient for moderate values of p, it is unfortunately unable to handle larger values: it is limited to $p \lesssim 2^{t/2}$ and becomes quite inefficient when p approaches this limit since it must use a small block size λ.

In **[45]**, we propose a method to overcome this limitation by using multiword arithmetic. We define

$$A_i = \left\lfloor \left(A - \sum_{k=0}^{i-1} \alpha^k A_k \right) / \alpha \right\rfloor, \qquad i = 0 \colon w_A - 1,$$
$$B_j = \left\lfloor \left(B - \sum_{k=0}^{j-1} \beta^k B_k \right) / \beta \right\rfloor, \qquad j = 0 \colon w_B - 1,$$

where the floor operator $\lfloor \cdot \rfloor$ is applied elementwise. This yields the decompositions

$$A = \sum_{i=0}^{w_A - 1} \alpha^i A_i$$
$$B = \sum_{j=0}^{w_B - 1} \beta^j B_j.$$

Note that, unlike in the previous section, the decompositions of A and B can use different numbers of words w_A and w_B, for reasons that will be clear shortly.

How should we choose the parameters α and β? The goal is to minimize the size of the largest coefficient in any A_i or B_j. Therefore, a suitable choice is $\alpha = \lceil p^{1/w_A} \rceil$ and $\beta = \lceil p^{1/w_B} \rceil$, which balances the size of the coefficients of each A_i and each B_j, ensuring that they are bounded by α and β, respectively.

We can then compute the modular product as

$$C = AB \bmod p = \sum_{i=0}^{w_A - 1} \sum_{j=0}^{w_B - 1} \alpha^i \beta^j (A_i B_j \bmod p) \bmod p, \tag{5.6}$$

where the partial products $T_{ij} = A_i B_j \bmod p$ can be efficiently computed block by block as in (5.5), and the scaled updates $C \leftarrow C + \alpha^i \beta^j T_{ij} \bmod p$ only require $O(w_A w_B)$ reductions per coefficient of C.

This multiword approach is summarized in Algorithm 5.2. The following result shows that it allows for handling much larger primes p.

Theorem 5.2 (Proposition 3.2 of **[45]**). *Let $C = AB \bmod p$ be evaluated with Algorithm 5.2 in floating-point arithmetic with t bits of significand. Under the condition*

$$p^{1/w_A + 1/w_B} \leq \frac{2^t}{\lambda}, \tag{5.7}$$

Algorithm 5.2 Multiword matrix multiplication over prime finite fields.

Input: Two matrices $A \in \mathbb{Z}^{m\times n}$ and $B \in \mathbb{Z}^{n\times q}$ reduced modulo p, the numbers of words w_A and w_B, a prime p, the block size λ.
Output: $C = AB \bmod p$ computed with (w_A, w_B)-word arithmetic.

$\alpha = \lceil p^{1/w_A} \rceil$, $\beta = \lceil p^{1/w_B} \rceil$
for $i = 0$ **to** $w_A - 1$ **do**
  $A_i = \left\lfloor (A - \sum_{k=0}^{i-1} \alpha^k A_k)/\alpha \right\rfloor$
end for
for $j = 0$ **to** $w_B - 1$ **do**
  $B_j = \left\lfloor (B - \sum_{k=0}^{j-1} \beta^k B_k)/\beta \right\rfloor$
end for
$C = 0$
for $i = 0$ **to** $w - 1$ **do**
  for $j = 0$ **to** $w - 1$ **do**
    Compute $T = A_i B_j \bmod p$ by blocks of size λ.
    $C = C + \alpha^i \beta^j T \bmod p$
  end for
end for

no overflow occurs and thus C is computed exactly.

Theorem 5.2 can be used to determine the necessary numbers of words (w_A, w_B) to handle a given prime size. Table 5.1 compares different choices for (w_A, w_B), with $w_A \leq w_B$ (the symmetric case being analogous). We report the maximum prime size that can be handled based on (5.7). We also report the limit on $\log_2 p$, that is, the maximum bitsize of p, when using double precision arithmetic ($t = 53$).

Table 5.1: Comparison between different (w_A, w_B) choices.

(w_A, w_B)	$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$	$(2,2)$	$(2,3)$
Number of products $(= w_A w_B)$	1	2	3	4	4	6
Limit on p	$2^{t/2}$	$2^{2t/3}$	$2^{3t/4}$	$2^{4t/5}$	2^t	2^t
Limit on $\log_2 p$ with $t = 53$	26	35	39	42	53	53
Maximum block size λ	$2^t/p^2$	$2^t/p^{3/2}$	$2^t/p^{4/3}$	$2^t/p^{5/4}$	$2^t/p$	$2^t/p^{5/6}$

Since the cost of each variant is proportional to the number of products $w_A w_B$, we can use this analysis to determine which variant to use depending on the prime size. The $(1,1)$ standard variant without multiword arithmetic requires a single product and so is the least expensive. It is thus expected to be the best choice as long as it can use a sufficiently large block size, that is, when $p \ll 2^{t/2}$. As p approaches this limit, the $(1,1)$ variant will become increasingly less efficient until it no longer is correct. Around this limit it will therefore become beneficial to switch to a multiword approach with the smallest possible cost, that is, $w_A = 1$ and $w_B = 2$; this $(1,2)$ variant should be the best until p approaches its new limit $p \ll 2^{2t/3}$. At this point, further increasing w_A and/or w_B will allow for handling larger p at the cost of additional products. The $(2,2)$ variant can handle any prime that can be stored in t bits of significand. Nevertheless, the $(2,3)$ variant is of interest when p approaches 2^t, because it can use a larger block size λ than the $(2,2)$ variant. This leads to more efficient products since it reduces the relative cost of

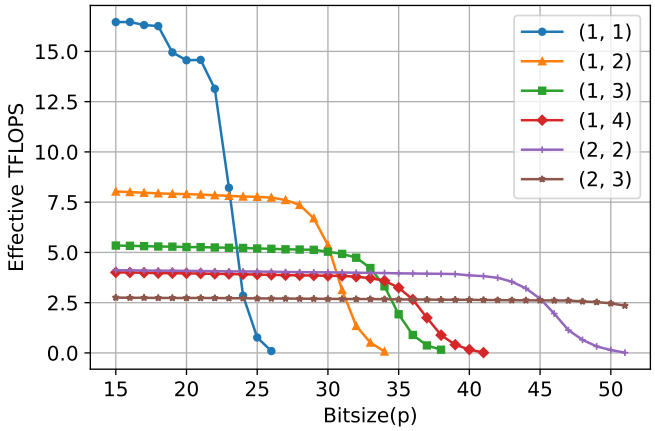


Figure 5.1: Performance of the matrix product $C = AB \bmod p$ where $A, B, C \in \mathbb{Z}^{n\times n}$ with $n = 10016$, depending on the bitsize t of p and on the (w_A, w_B) variant of Algorithm 5.2 used. The performance is measured in effective TFLOPS, that is, the inverse of time scaled by $2n^3 \times 10^{-12}$. The benchmark is run on a A100 NVIDA GPU.

the reductions and also increases the arithmetic intensity of the matrix products. We validate this theoretical comparison experimentally in Figure 5.1, which reports the performance of each (w_A, w_B) variant depending on the bitsize t of p. This benchmark uses square matrices and is run on a A100 NVIDIA GPU; see **[45]** for additional performance benchmarks on various matrix shapes and on both CPU and GPU architectures. In particular, for unbalanced matrix shapes, we propose (see Algorithm 3.3 in **[45]**) to concatenate the words (for example, $B_{\text{conc}} = [B_1 \cdots B_{w_B}]$) and to compute $A_i B_{\text{conc}}$ to increase the arithmetic intensity of the products.

It is worth noting that there exists other approaches to handle primes $p > 2^{t/2}$. For example, arbitrary precision arithmetic can, by design, handle any prime size. However, it is very costly, and cannot exploit the standard BLAS routines that make our approach highly efficient; it is thus not very suited for intermediate prime sizes such as $2^{t/2} < p \leq 2^t$. A more serious competitor is multimodular arithmetic, which is based on the Chinese remainder theorem. We have already mentioned the Ozaki scheme II [218] for emulating high precision floating-point arithmetic. A similar approach can be used for computing $AB \bmod p$ with $p > 2^{t/2}$, and is for example described by Doliskani et al. [114]. However, this approach requires

$$\left\lceil \frac{4f + 2d}{t - \ell} \right\rceil$$

products, where $f = \log_2 p$, $d = \log_2 k$, and $\ell = \log_2 \lambda$. In comparison, by Theorem 5.2, our multiword approach only requires $p^{1/w_A + 1/w_B} \leq 2^t/\lambda$, which leads to a number of products

$$w_A w_B = \left\lceil \frac{(w_A + w_B)f}{t - \ell} \right\rceil.$$

Since we can expect to have $w_A + w_B \leq 4$ for almost all primes fitting on t bits, our multiword approach will almost always require fewer products than a multimodular approach. It is worth noting that this conclusion might change when considering mixed precision matrix multiply–accumulate hardware (see Research Problem 16.5).

Chapter 6

Low-rank approximations

Low-rank approximations (LRA) are a powerful tool to reduce the dimensionality of large scale data. Given a dense matrix $A \in \mathbb{R}^{m\times n}$ with $m \geq n$, we seek to approximate it by a rank-k product $A \approx XY^T$, where $X \in \mathbb{R}^{m\times k}$ and $Y \in \mathbb{R}^{n\times k}$. Naturally, the low-rank case ($k \ll n$) is of interest since, in this case, storing the low-rank factors X and Y requires fewer entries than storing the full matrix A, and computations involving A will usually also require fewer flops.

For any unitarily invariant norm, the optimal LRA is the truncated singular value decomposition (SVD), a result known as the Eckart–Young theorem [124]. Specifically, given the SVD $A = U\Sigma V^T$, the optimal rank-k approximation of A is

$$\arg \min_{\substack{M\in\mathbb{R}^{m\times n} \\ \text{rank}(M)=k}} \|A - M\| = U_k\Sigma_k V_k^T,$$

where $U_k\Sigma_k V_k^T$ is the truncated SVD of A, formed by the first k singular vectors and values only. However, computing the truncated SVD requires computing first the full SVD, which requires $O(mn^2)$ operations, and so in practice this optimal method is too expensive and commonly replaced with suboptimally accurate, but much cheaper methods.

In this chapter, we review some of these methods, as well as techniques to accelerate them via mixed precision and/or randomization.

The contributions described in this chapter are based on the following papers: **[16]**, **[21]** for Section 6.3; **[22]**, **[33]** for Section 6.5; **[9]** for Section 6.6; and **[42]** for Section 6.7.

6.1 QR with column pivoting (QRCP)

A QR factorization capable of revealing the rank of a matrix can be obtained using column pivoting as in the method proposed by Businger and Golub [82]. Essentially, at each step k of the QR factorization, this method permutes the columns of the trailing submatrix such that the pivotal column k is the one that has maximum 2-norm. This implies that the diagonal coefficients of the R factor are of decreasing absolute value and that the following property holds:

$$AP = QR \text{ where } |R_{k,k}| \geq \|R_{k:m,j}\|_2 \quad j = k, ..., n. \tag{6.1}$$

Note that explicitly recomputing the norm of all the columns in the trailing submatrix not only is expensive but completely prevents the use of blocking (and, thus, of BLAS-3 operations) because this requires to entirely update all the remaining columns after every elimination step. This problem can be partially overcome by taking advantage of the fact that, once all the column norms of the original A matrix have been computed, these do not have to be recomputed at every

step but can be cheaply updated. Essentially, at step k of the factorization, the norm of column j in the trailing submatrix is updated by subtracting the freshly computed $R_{k,j}$ coefficient. This approach is shown in Algorithm 6.1, which we refer to as QRCP; here we have assumed that V is a matrix containing all the computed v^k vectors in its columns, that householder(x, k) is a function which computes and applies a Householder reflection that annihilates the bottom $m-k+1$ coefficients of a vector x of size m and that W is a workspace. Although, in this algorithm, a large portion of computations is still done using BLAS-2 operations, some BLAS-3 can be used (in line 17).

Algorithm 6.1 Blocked QR factorization with column pivoting (QRCP).

Input: $A \in \mathbb{R}^{m\times n}$, $\varepsilon > 0$.
Output: a matrix with orthonormal columns $Q \in \mathbb{R}^{m\times k}$, an upper triangular matrix $R \in \mathbb{R}^{k\times n}$, and a permutation matrix $P \in \mathbb{R}^{n\times n}$ such that $\|AP - QR\| \leq \varepsilon\|A\|$.

1: $P = I$
2: $\eta_j = \|A_{:j}\|_2, \quad j = 1, \ldots, n$
3: **for** $j = 1 : b : n$ **do**
4: **for** $k = j : j + b - 1$ **do**
5: **if** $\|\eta_{k:n}\|_2 \leq \varepsilon$ **then**
6: $b = k - j; k = k - 1$
7: **goto** 17 and **break** j loop
8: **end if**
9: Find $i \in k, \ldots, n$ such that η_i is minimal
10: Swap columns k and i in A and P
11: $A_{k:m,k} = A_{k:m,k} - V_{k:m,j:k-1}W_{j:k-1,k}$
12: $v^k, \ \tau^k = \text{householder}\,(A_{:,k}, \ k)$
13: $W_{k+1,k:n} = \tau^k (v^k)^T A_{:,k+1:n} + \tau^k (v^k)^T V_{:,j:k-1} W_{j:k-1,k+1:n}$
14: $A_{k,k+1:n} = A_{k,k+1:n} - V_{k,j:k-1}W_{j:k-1,k+1:n}$
15: $\eta_i = \sqrt{\eta_i^2 - A_{k,i}^2}, \ i = k+1, ..., n$
16: **end for**
17: $A_{j+b:m,j+b:n} = A_{j+b:m,j+b:n} - V_{j+b:m,j:j+b-1}W_{j:j+b-1:j+b:n}$
18: **end for**
19: $Q = \prod_{k=1,n} (I - \tau^k v^k (v^k)^T)$, $Q = Q_{:,1:k}$
20: $R = \text{triu}(A)$, $R = R_{1:k,:}$

In practice, Algorithm 6.1 achieves poor performance and parallel scalability because a large portion of computations in Algorithm 6.1 are of BLAS-2 type and because of the numerous communications that the Businger-Golub pivoting requires. For this reason alternative pivoting techniques have been proposed in the literature that aim at overcoming these drawbacks. Multiple methods proposed in the literature, such as those by [117, 195, 110, 111], rely on an approach where, at every step of the factorization, not one but b (the panel size) selected columns of the trailing submatrix are moved upfront, eliminated and the corresponding transformations applied at once using BLAS-3 operations. These methods essentially differ in the way these b columns are selected. In the approach proposed by Duersch and Gu [117] and by Martinsson et al. [195], outlined in Algorithm 6.2, this selection is done using randomized sampling, that is, the trailing submatrix is left-multiplied by a Gaussian matrix $\Omega \in \mathcal{N}(0,1)^{(b+p)\times m}$ which produces a sample matrix S; here p is a small *oversampling* parameter. Because of the properties that connect S to the trailing submatrix, the "important" b columns can be selected by applying QRCP to the sample matrix S. This drastically improves the amount of BLAS-3 operations because S has a

much smaller row-dimension than the trailing submatrix. As a matter of fact, the sample matrix S does not have to be recomputed at every factorization step but it can be computed only once and cheaply updated [117, 195].

Algorithm 6.2 Blocked QR factorization with randomized pivoting (QRRP).

Input: $A \in \mathbb{R}^{m\times n}$, $\varepsilon > 0$.
Output: a matrix with orthonormal columns $Q \in \mathbb{R}^{m\times k}$, an upper triangular matrix $R \in \mathbb{R}^{k\times n}$, and a permutation matrix $P \in \mathbb{R}^{n\times n}$ such that $\|AP - QR\| \leq \varepsilon\|A\|$.

$P = I$
Draw a random Gaussian matrix $\Omega \in \mathcal{N}(0,1)^{(b+p)\times m}$
$S = \Omega A$
for $j = 1 : b : n$ **do**
  $\tilde{Q}, \tilde{R}, \tilde{P} = \mathrm{QRCP}(S_{:,j:n}, \varepsilon)$
  $A_{:,j:n} = A_{:,j:n}\tilde{P},\ P_{:,j:n} = P_{:,j:n}\tilde{P}$
  $Q^j,\ R^j = QR(A_{j:m,j:j+b-1})$
  $A_{j:m,j+b:n} = (Q^j)^T A_{j:m,j+b:n}$
  Update S
end for
$Q = \prod_j Q^j$, $Q = Q_{:,1:k}$
$R = \mathrm{triu}(A)$, $R = R_{1:k,:}$

Xiao et al. [253] demonstrate that the property of equation (6.1) does not apply formally to the result of QRRP but holds in a probabilistic sense. Given $\varepsilon,\ \Delta \in (0,1)$ and an oversampling parameter $p \geq \lceil \frac{4}{\varepsilon^2-\varepsilon^3}\log(\frac{2nk}{\Delta})\rceil$ the following property

$$|R_{k,k}| \geq \sqrt{\frac{1-\varepsilon}{1+\varepsilon}}\|R_{k:m,j}\|_2, \quad i+1 \leq j \leq n \tag{6.2}$$

holds with probability at least $1-\Delta$.

6.2 Randomized range finder

Another popular class of randomized approaches to compute LRA are based on range finder methods; see the survey of Halko et al. [154]. They consist in constructing a (usually orthonormal) matrix $Q \in \mathbb{R}^{m\times\ell}$ that is an approximate basis of the range of $A \in \mathbb{R}^{m\times n}$, that is, a matrix Q such that $A \approx QQ^TA$. This basis Q can be efficiently computed via random projections $A\Omega$ where Ω is a Gaussian matrix. The simplest scenario is when the target rank k is fixed. In this case, it suffices to set $\ell = k+p$, where p is a small oversampling parameter, and compute $Q = \texttt{orth}(A\Omega)$ with $\Omega \in \mathbb{R}^{n\times\ell}$. This yields an approximate basis Q satisfying [154, Theorem 10.5]

$$\mathbb{E}(\|A - QQ^TA\|_F) \leq \left(1 + \frac{k}{p-1}\right)^{1/2}\left(\sum_{i>k}\sigma_i^2\right)^{1/2}. \tag{6.3}$$

The term $(\sum_{i>k}\sigma_i^2)^{1/2}$ shows that the average error corresponds to a rank-k approximation, up to the constant $(1+k/(p-1))^{1/2}$ that can be made close to 1 by increasing the oversampling p. Thus, the relation $A \approx QQ^TA$ yields a rank-ℓ approximation of rank-k quality; a rank-k approximation XY^T can then be obtained by forming $B = Q^TA$, computing a rank-k approximation $B \approx \bar{X}Y^T$ (via any deterministic LRA method), and finally setting $X = Q\bar{X}$. This fixed-rank range finder

method is summarized in Algorithm 6.3. The cost of the deterministic LRA method is much lower than if applied to the original matrix A, since $B \in \mathbb{R}^{\ell \times n}$ is of smaller dimensions. In fact, the asymptotic complexity of the algorithm is dominated by the matrix–matrix products $S = A\Omega$ and $B = Q^T A$, which are very efficient. It is worth noting that, if ℓ is not much larger than k, and if it is not critical to obtain the lowest possible rank, the cost of the algorithm may be reduced by skipping the last steps and simply using the rank-ℓ approximation $A \approx QB$.

A more complex but often realistic scenario is to fix the target accuracy ε and adaptively compute the required size of Q such that $\|A - QQ^T A\| \leq \varepsilon \|A\|$. The key idea is to build Q incrementally and stop whenever the target accuracy is satisfied. Several such fixed-accuracy range finder methods have been proposed, mainly depending on how the stopping criterion is implemented. For example, Halko et al. [154, sect. 4.4] propose a clever method that estimates the norm of $A - Q_i Q_i^T A$ at step i using the norm of $(A - Q_i Q_i^T A)\Omega_{i+1}$, which is exactly what needs to be computed at step $i+1$ if the method is not stopped at step i. Martinsson and Voronin [196] proposed a more direct method, outlined in Algorithm 6.4, that explicitly computes the error by building the matrix $A - Q_i Q_i^T A = A - Q_i B_i$. This method requires one extra matrix–matrix product but provides a more reliable stopping criterion (see Research Problem 16.6).

Algorithm 6.3 Fixed-rank randomized range finder.

Input: $A \in \mathbb{R}^{m \times n}$, the target rank k.
Output: $X \in \mathbb{R}^{m \times k}$, $Y \in \mathbb{R}^{n \times k}$ such that $A \approx XY^T$

1: Draw a random Gaussian matrix $\Omega \in \mathbb{R}^{n \times k+p}$.
2: $S = A\Omega$
3: $Q = \texttt{orth}(S)$
4: $B = Q^T A$
5: $\bar{X}Y^T = \texttt{LRA}(B, k)$
6: $X = Q\bar{X}$

Algorithm 6.4 Fixed-accuracy randomized range finder (Martinsson and Voronin [196] variant).

Input: $A \in \mathbb{R}^{m \times n}$, the target accuracy ε.
Output: $X \in \mathbb{R}^{m \times k}$, $Y \in \mathbb{R}^{n \times k}$ such that $\|A - XY^T\| \leq \varepsilon \|A\|$.

1: Initialize Q and B to empty matrices.
2: $\eta_A = \|A\|$
3: **repeat**
4: Draw a random Gaussian matrix $\Omega \in \mathbb{R}^{n \times b}$.
5: $S = A\Omega$
6: $Q_b = \texttt{orth}(S - Q(Q^T S))$
7: $B_b = Q_b^T A$
8: $Q \leftarrow [Q\ Q_b]$
9: $B \leftarrow \begin{bmatrix} B \\ B_b \end{bmatrix}$
10: $A \leftarrow A - Q_b B_b$
11: **until** $\|A\| \leq \varepsilon \eta_A$
12: $\bar{X}Y^T = \texttt{LRA}(B, \varepsilon)$
13: $X = Q\bar{X}$

6.3 Adaptive precision LRA

A first opportunity to exploit mixed precision arithmetic for computing low-rank matrices arises when the matrix exhibits rapidly decaying singular values. Consider the SVD $A = U\Sigma V^T$ and the LRA $\bar{U}\bar{\Sigma}\bar{V}^T$ obtained by truncating the smallest singular values such that $\|A - \bar{U}\bar{\Sigma}\bar{V}^T\| \leq \varepsilon\|A\|$. In what precision should the LRA $\bar{U}\bar{\Sigma}\bar{V}^T$ be stored in? Clearly, if we wish to maintain an accuracy of order ε, the answer can only be a precision with a unit roundoff safely smaller than ε.

In **[16]**, we make the key observation that singular vectors associated with the smaller singular values (among those that have been kept) do not need to be stored with as much accuracy as those associated with the larger singular values. Assume p precision formats are available with unit roundoffs $u_1 \leq \varepsilon < u_2 < \ldots < u_p$ (we only need the first precision to provide an accuracy of at least ε, and the remaining $p-1$ precisions are the "low" precisions). We define a blockwise partitioning

$$\bar{U}\bar{\Sigma}\bar{V}^T = [U_1 \ldots U_p] \begin{bmatrix} \Sigma_1 & & \\ & \ddots & \\ & & \Sigma_p \end{bmatrix} [V_1 \ldots V_p]^T \tag{6.4}$$

and let

$$\widehat{\bar{U}}\widehat{\bar{\Sigma}}\widehat{\bar{V}}^T = \begin{bmatrix} \widehat{U}_1 \ldots \widehat{U}_p \end{bmatrix} \begin{bmatrix} \widehat{\Sigma}_1 & & \\ & \ddots & \\ & & \widehat{\Sigma}_p \end{bmatrix} \begin{bmatrix} \widehat{V}_1 \ldots \widehat{V}_p \end{bmatrix}^T \tag{6.5}$$

where $\widehat{U}_i$, $\widehat{\Sigma}_i$, and $\widehat{V}_i$ are stored in the ith precision.

The following result determines a rigorous criterion to build this partitioning while preserving an accuracy of order ε.

Theorem 6.1 (Theorem 2.1 of **[16]**). *If the partitioning* (6.4) *satisfies* $\|\Sigma_i\| \leq \varepsilon\|A\|/u_i$, *then*

$$\|A - \widehat{\bar{U}}\widehat{\bar{\Sigma}}\widehat{\bar{V}}^T\| \leq (2p-1)\varepsilon\|A\|.$$

The criterion $\|\Sigma_i\| \leq \varepsilon\|A\|/u_i$ shows that the singular values and their associated singular vectors can be stored in a precision that is inversely proportional to their magnitude. Hence, if the singular values of A decay rapidly, the representation (6.5) allows for a significant use of low precisions.

It is important to note that this approach is not only applicable to the SVD, but also to other LRA representations provided that they can also be split into rank-one components that gradually decrease in norm (Lemma 2.2 of **[16]**): this is notably the case of the QRCP decomposition.

The adaptive precision representation (6.5) can be used to reduce the storage cost of the LRA, and the cost of any subsequent operation using that LRA; we will notably see this in the case of the BLR LU factorization in Section 9.4. There remains the question of whether we can also reduce the cost of computing the LRA itself: that is, rather than computing the LRA in high precision and then switching it to adaptive precision, can we directly compute its adaptive precision representation? While the answer naturally depends on the specific LRA method that is considered, it is positive for at least three methods: QRCP (Algorithm 6.1), its variant with randomized pivoting (Algorithm 6.2), and the fixed-accuracy randomized range finder (Algorithm 6.4). The key idea in all three cases is to start the algorithm in the highest precision u_1, and to switch to a lower precision u_k whenever the norm of the approximation error falls below $\varepsilon\|A\|/u_k$. In the case of Algorithm 6.4, this is straightforward since the error is

explicitly computed by deflating A. This observation is leveraged by Connolly et al. [100], who propose a mixed precision version of Algorithm 6.4; see [100, Algorithm 3.2].

In the case of a QR factorization, the situation is quite similar. In **[21]**, we prove the following result.

Theorem 6.2 (Theorem 3.3 of **[21]**). *Assume that a truncated QR factorization is computed such that $k \leq n$ Householder transformations are computed and applied to a matrix $A \in \mathbb{R}^{m\times n}$ using p different precisions with increasing unit roundoffs $u_1 < \ldots < u_p$. Let k_i be the number of transformations that are computed in precision u_i and let A_i be the trailing submatrix after $\sum_{j=1}^{i-1} k_j$ transformations are applied. The computed $\widehat{R}_i$ and $\widehat{Q}_i$ satisfy*

$$\|A - \sum_{i=1}^{p} \widehat{Q}_i \widehat{R}_i\|_F \leq \|A_{p+1}\|_F + \sum_{i=1}^{p} \sqrt{k_i}\widetilde{\gamma}^{(i)}_{mk_i}\|A_i\|_F, \tag{6.6}$$

where $\widetilde{\gamma}^{(i)}_{mk_i} = cmk_iu_i/(1 - cmk_iu_i)$ and c is a small absolute constant.

Theorem 6.2 shows that the precision switch should be guided by the norm of the trailing submatrix: indeed, by (6.6) an error of order ε can be achieved by truncating so that $\|A_{p+1}\| \leq \varepsilon$ and by switching precisions so that $u_i\|A_i\| \leq \varepsilon$ for all $i = 1\colon p$. Note that Theorem 6.2 holds regardless of how pivoting is performed; in fact, it works even without pivoting. Obviously, the use of pivoting leads to a faster decay of the trailing submatrix norm and, consequently, to an earlier truncation and change of precisions.

The pivoting method does affect, however, the way the norm of the trailing submatrix $\|A_i\|$ is estimated. In the case of a standard QRCP factorization (Algorithm 6.1), this norm can be efficiently computed from column norms η that are kept updated as the factorization progresses. In the case of randomized pivoting (Algorithm 6.2), the norm of the trailing submatrix cannot be easily computed. Nevertheless, it is possible to check for precision switch and truncation within the QRCP factorization of the sample matrix based on the norm of the trailing submatrix of the sample. Indeed, according to [117, Theorem 3.1], after i transformations, the square norm of the columns in the trailing submatrix of the sample can be written as a factor x_i of the square norm of the actual trailing columns, where x_i follows the truncated χ^2 distribution with $b+p-i$ degrees of freedom. A conservative bound for all $i \leq b$ is therefore $x_i \leq p$: in our experiments of **[21]**, we have confirmed that replacing ε by $\sqrt{p}\varepsilon$ works indeed well on a wide range of problems.

We illustrate the potential of this adaptive precision QR factorization in Figure 6.1 with an application to image compression. The figure compares the original image (top left) with images that are compressed using a truncated QR factorization in uniform fp32 (top right) or bfloat16 (bottom right) precision, or in adaptive fp32/bloat16 precision (bottom-left). Here, the truncation threshold ε was set to 0.04 and the truncation happened on column 191 in all three cases. Clearly, using only bfloat16 does not lead to satisfactory compression. Both the other two approaches, instead, achieve a satisfactory result although, in the adaptive precision case, only 12 out of 190 columns are computed and stored in fp32 and the rest in bfloat16.

6.4 Multiword arithmetic for LRA

Another natural idea to exploit mixed precision in LRA is to rely on multiword arithmetic (Chapter 5). Indeed, multiword matrix multiplication is very efficient on mixed precision GPUs, and many LRA methods heavily rely on matrix multiplication. Taking the fixed-rank randomized range finder as an example, Algorithm 6.3 requires three steps, the two most expensive of which are matrix products: they require $O(mnk)$ flops, whereas the truncated QRCP step only requires

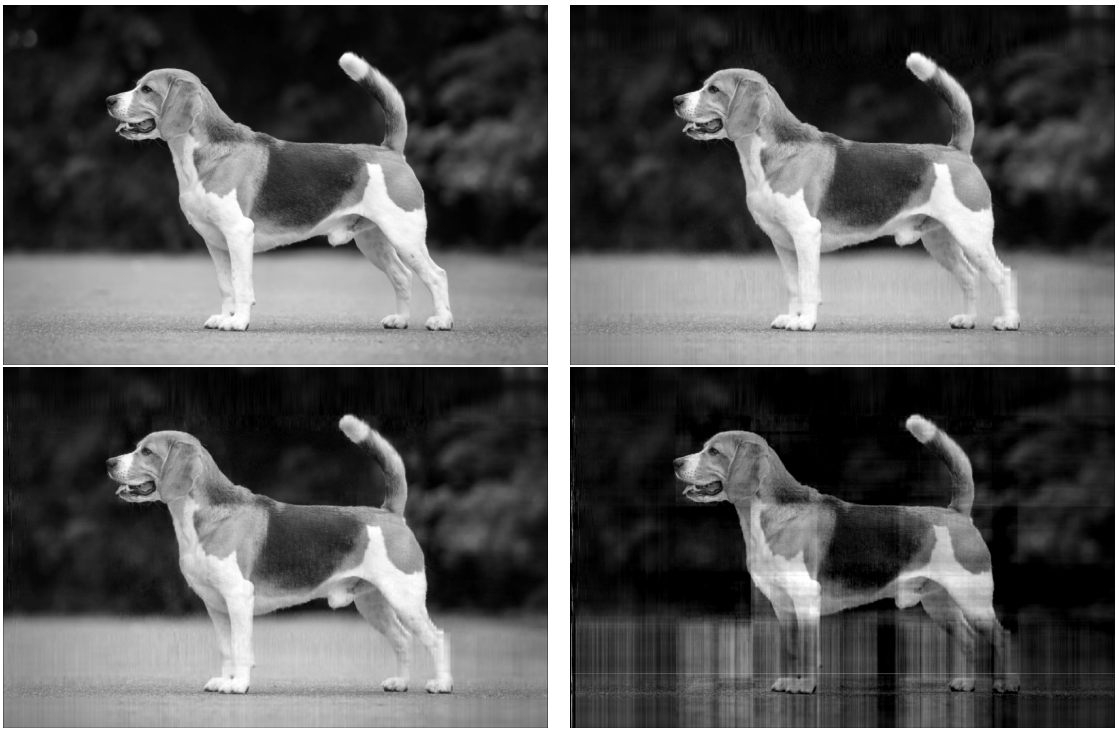


Figure 6.1: Image compression. Original image (top left); reconstructed image after compression in full fp32 (top right), in fp32/bfloat16 (bottom left) and full bfloat16 (bottom right). ε was set to 0.04 and the truncation happened at column 191 but in the mixed-precision case only 12 transformations were computed in fp32 and the rest in bfloat16. Image size is 1057×1600.

$O(mk^2)$ flops. Therefore for large matrices we may expect LRA to be accelerated by multiword arithmetic by the same speedup than matrix multiplication.

In fact, Ootomo and Yokota [212] show that in the specific case of randomized methods we can do even better by reducing the cost of the $A\Omega$ matrix product. Considering for example double-fp16 arithmetic, the standard approach described in Section 5.2 would decompose $A \approx A_0 + A_1$ and $\Omega \approx \Omega_0 + \Omega_1$, where each word is stored in fp16, and would then compute $A\Omega \approx A_0\Omega_0 + A_1\Omega_0 + A_0\Omega_1$ via three matrix products. However, in this case, Ω is a random Gaussian matrix, so it is natural to ask how accurately we really need to represent it. Ootomo and Yokota [212] provide an answer by showing that storing a random Gaussian matrix in fp16 arithmetic only slightly increases its variance, and that this increase preserves the range finding properties of the algorithm. Therefore, we can compute $A\Omega$ via only two matrix products $A_0\Omega + A_1\Omega$ where Ω is stored in fp16. This observation yields Algorithm 6.5, which can achieve fp32 accuracy while only requiring five matrix products with fp16/fp32 tensor cores. We note that the method proposed in [212] only uses multiword arithmetic for the product $A\Omega$, but not for A^TX.

6.5 Iterative refinement for LRA

In **[22]**, we propose yet another approach to compute LRA in mixed precision which draws inspiration from iterative refinement (Chapter 10). Given a linear system $Ax = b$ and a low precision solution x_0, iterative refinement consists in computing the residual $r = b - Ax_0$ in high precision, solving the linear system $Ad = r$ in low precision, and updating the solution as

Algorithm 6.5 Fixed-rank randomized range finder with double-fp16 arithmetic.

Input: $A \in \mathbb{R}^{m\times n}$, the target rank k.
Output: $X \in \mathbb{R}^{m\times k}$, $Y \in \mathbb{R}^{n\times k}$ such that $A \approx XY^T$

1: Draw a random Gaussian matrix $\Omega \in \mathbb{R}^{n\times k+p}$ in fp16.
2: $A_0 = \text{fp16}(A)$ and $A_1 = \text{fp16}(A - A_0)$
3: $B = A_0\Omega + A_1\Omega$ with fp16/fp32 tensor cores
4: $X = \texttt{TQRCP}(B)$ in fp32
5: $X_0 = \text{fp16}(X)$ and $X_1 = \text{fp16}(X - X_0)$
6: $Y = A_0^T X_0 + A_0^T X_1 + A_1^T X_0$ with fp16/fp32 tensor cores

$x_1 = x_0 + d$. Transposing this approach to the LRA problem, we obtain the method outlined in Algorithm 6.6.

The method first computes a low precision LRA $A \approx XY^T$. Then, it computes the error $E = A - XY^T$ in high precision and compresses it via another low precision LRA $E \approx X_E Y_E^T$. It finally updates the LRA as $XY^T = XY^T + X_E Y_E^T = [X\ X_E][Y\ Y_E]^T$. This method is based on the key observation that if A is a numerically low-rank matrix, then the error matrix $E = A - XY^T$ is also numerically low rank, with a rank bounded by at most twice the rank of A. Therefore, the refined LRA $[X\ X_E][Y\ Y_E]^T$ has a rank at most three times larger than the rank of A at the same accuracy. To prevent the rank from growing uncontrollably, Algorithm 6.6 recompresses the LRA at each iteration via an LRA on $[X\ X_E][Y\ Y_E]^T$. This recompression must be performed in high precision but can exploit the low rank of its input to be much less expensive than an LRA on a full matrix.

Algorithm 6.6 Iterative refinement for LRA.

Input: the matrix A, the number of iterations n_{it}, a given LRA method `LRA`, and four precision parameters $\varepsilon_{\text{low}} < u_{\text{low}} \ll \varepsilon_{\text{high}} < u_{\text{high}}$.
Output: XY^T such that $\|A - XY^T\| \le \varepsilon_{\text{high}}\|A\|$.

1: Compress $XY^T = \texttt{LRA}(X, \varepsilon_{\text{low}})$ in precision u_{low}.
2: **for** $i = 1$ **to** n_{it} **do**
3: Compute $E = A - XY^T$ in precision u_{high}.
4: Compress $X_E Y_E^T = \texttt{LRA}(E, \varepsilon_{\text{low}})$ in precision u_{low}.
5: Recompress $XY^T = \texttt{LRA}([X\ X_E][Y\ Y_E]^T, \varepsilon_{\text{high}})$ in precision u_{high}.
6: **end for**

Algorithm 6.6 has four parameters that control its accuracy: ε_{low}, $\varepsilon_{\text{high}}$, u_{low}, and u_{high}. The "ε" parameters correspond to the low-rank truncation thresholds used by the LRA, whereas the "u" parameters correspond to the unit roundoffs of the arithmetics. In order for an LRA to reliably detect the numerical rank at a given threshold ε, it needs to be performed in an arithmetic with a unit roundoff u safely smaller than ε. Thus, we have $\varepsilon_{\text{low}} < u_{\text{low}} \ll \varepsilon_{\text{high}} < u_{\text{high}}$. The following result characterizes the convergence behavior of Algorithm 6.6.

Theorem 6.3 (Theorem 2.1 of [22]). *Let Algorithm 6.6 be applied to A with an `LRA` method satisfying $\|A - \texttt{LRA}(A,\varepsilon)\| \le (\varepsilon + cu)\|A\|$ for any truncation threshold ε and unit roundoff u. Then after n_{it} iterations, the computed factors XY^T satisfy*

$$\|A - XY^T\| \le c\big(\varepsilon_{\text{low}}^{n_{\text{it}}+1} + \varepsilon_{\text{high}} + O(\varepsilon_{\text{low}}\varepsilon_{\text{high}})\big)\|A\|, \tag{6.7}$$

where c is a constant only depending on the dimensions of the matrix.

The bound (6.7) shows that the error is reduced at each iteration by a factor ε_{low}, which thus controls the convergence speed, until the error reaches $\varepsilon_{\text{high}}$, the attainable accuracy. This means that we can actually estimate in advance how many iterations are needed to achieve the desired accuracy $\varepsilon_{\text{high}}$:

$$n_{\text{it}} = \lceil \log(\varepsilon_{\text{high}}) / \log(\varepsilon_{\text{low}}) \rceil - 1, \tag{6.8}$$

which explains why Algorithm 6.6 does not need a stopping criterion.

It is worth noting that, unlike iterative refinement for linear systems (Chapter 10), there is no dependence on the condition number of A in (6.7) and thus Algorithm 6.6 is always guaranteed to converge as long as $\varepsilon_{\text{low}} < 1$. This is explained by the fact that the speed of convergence depends on the backward error $\|A - XY^T\|$, rather than the forward one. Therefore, Algorithm 6.6 is extremely general: it can be applied to any matrix of low numerical rank and it can make use of potentially very low precisions.

From a numerical perspective, Algorithm 6.6 is therefore quite appealing. It now only remains to determine under which conditions it is also attractive from a computational perspective. Indeed, even if we neglect the recompression step whose cost should be asymptotically negligible, the algorithm still requires $n_{\text{it}}+1$ LRAs and n_{it} matrix multiplications. We perform a cost analysis in **[22]** which identifies two key situations where Algorithm 6.6 can significantly outperform a standard LRA in uniform precision. Interestingly, these two situations share strong connections with the two previous sections, Section 6.3 and Section 6.4, respectively.

- The first situation is when the numerical rank of the matrix is small at low accuracy levels, that is, when its singular values decay rapidly. This leads indeed to LRAs of lower rank and thus lower cost at the early iterations of Algorithm 6.6. Here, this iterative refinement approach can be seen as an adaptive precision one (Section 6.3): the large singular values will be first computed in low precision in the early iterations and then refined to high accuracy in the later ones, whereas the small singular values will only appear in the later iterations and be computed in both low precision and low accuracy.

- The second situation is when using a specialized mixed precision hardware which provides both a low precision arithmetic that is much faster than the high precision one, and also the ability to accumulate matrix products in high precision. This is notably the case of GPU tensor cores as discussed in Section 4.1. In this case, Algorithm 6.6 can perform not only the LRAs, but also the matrix products to compute E, using GPU tensor cores. Thus, it can achieve significant speedups even if it requires more flops than a standard LRA. An analogy can therefore be made with multiword arithmetic (Section 6.4), which also yields speedups despite performing more flops by exploiting fast hardware. In fact, the connection with multiword arithmetic goes beyond this simple observation. Given a rank-k LRA XY^T, one could define its two-word decomposition as $(X_0 + X_1)(Y_0 + Y_1)^T$, where the X_i and Y_i are stored in low precision. By developing the product and dropping the second-order term $X_1Y_1^T$, this expression can be rewritten as

  $$X_0Y_0^T + X_0Y_1^T + X_1Y_0^T = [X_0\ X_0\ X_1][Y_0\ Y_1\ Y_0]^T,$$

  which is a rank-$3k$ LRA. In fact, this is exactly the LRA that one step of iterative refinement would produce.

In **[33]**, we confirm the potential of this iterative refinement approach by implementing it on GPU tensor cores, using fp32 as the working precision and fp16 as the low precision. We consider the fixed-rank randomized range finder (Algorithm 6.3) as LRA. This choice of LRA is motivated by the fact that its bottleneck lies in matrix–matrix products, which are very efficient on GPUs and with fp16 tensor cores especially. In fact, these products are accelerated by

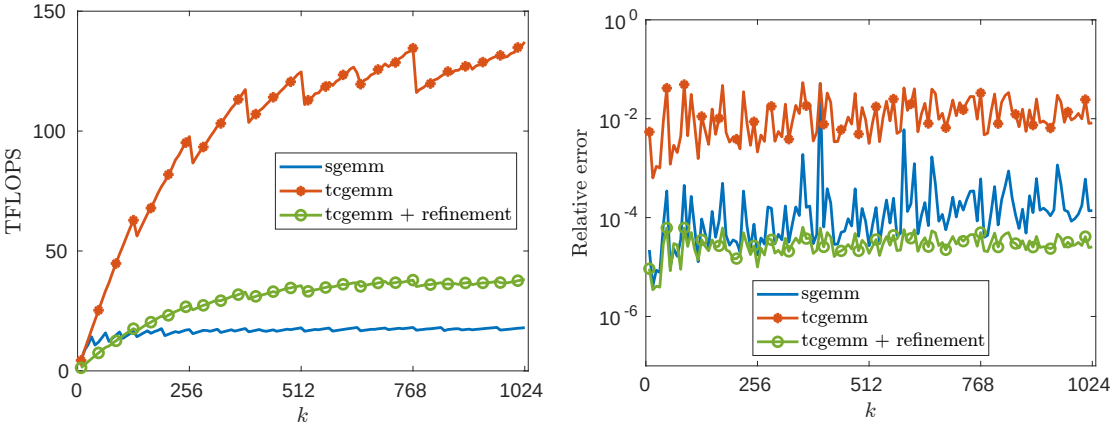


Figure 6.2: Performance and accuracy of fixed-k randomized range finder (Algorithm 6.3), depending on the precision of the matrix products: standard fp32 (`sgemm`) or with fp16 tensor cores (`tcgemm`). The variant with refinement uses Algorithm 6.6.

such a large factor that the orthonormalization of the sample S, when performed via a standard Householder QR factorization, becomes the new bottleneck, despite requiring an asymptotically negligible number of flops. This performance limitation can be overcome by switching to a Cholesky QR factorization, which is more suited for GPU architectures. While the loss of orthogonality with Cholesky QR is proportional to $\kappa(S)^2$, various methods have been proposed to (partially) mitigate its instability: reorthonormalization [133], shifting [131], pivoting [132], mixed precision [256], replacing the Cholesky factorization by an SVD [238, 257], preconditioning [240], randomization [134, 197], etc. In our context, the simplest and most efficient solution is actually to increase the precision of Cholesky QR from fp32 to fp64, which is sufficient to prevent instabilities, and almost does not degrade performance, because the NVIDIA A100 GPUs used for this study provide fp64 tensor cores that achieve the same speed as standard fp32 arithmetic.

In Figure 6.2, we compare the performance and accuracy of our implementation of the randomized range finder using matrix products on fp16/fp32 tensor cores (`tcgemm`) with the same algorithm using standard fp32 products (`sgemm`). achieves an average accuracy of order 10^{-4}. The figure shows that using `tcgemm` is up to 8× faster, but with an average accuracy of order 10^{-2} instead of 10^{-4} with `sgemm`. In order to recover the accuracy, we implement iterative refinement (Algorithm 6.6); we show that a single step of refinement is sufficient to significantly improve the accuracy to an average of order 10^{-5}, while remaining up to 2.2× faster than the standard `sgemm` version. This work illustrates the convergence of approximate computing techniques by combining low-rank approximations, randomization, mixed precision arithmetic, and GPU acceleration.

6.6 Gram LRA

Given $A \in \mathbb{R}^{m\times n}$ and its SVD $A = U\Sigma V^T$, the eigenvalue decomposition (EVD) of the Gram matrix $G = A^TA$ is $V\Sigma^2V^T$. This basic observation provides a simple method to compute the truncated SVD of A from the truncated EVD of G, outlined in Algorithm 6.7.

This Gram LRA approach is computationally attractive when $m \gg n$ because its bottleneck lies in the matrix–matrix product A^TA, which requires $O(mn^2)$ flops, whereas the EVD of G only requires $O(n^3)$ flops.

Algorithm 6.7 Gram low-rank approximation.

Input: $A \in \mathbb{R}^{m\times n}$, the target accuracy ε.
Output: $X \in \mathbb{R}^{m\times k}$, $Y \in \mathbb{R}^{n\times k}$ such that $\|A - XY^T\| \leq \varepsilon\|A\|$.

1: Compute the Gram matrix $G = A^T A \in \mathbb{R}^{n\times n}$.
2: Compute the EVD $G = W\Lambda W^T$.
3: Truncate W and Λ into $\bar{W} \in \mathbb{R}^{n\times k}$ and $\bar{\Lambda} \in \mathbb{R}^{k\times k}$, where k is the smallest integer such that $\|\Lambda - \bar{\Lambda}\| \leq \varepsilon^2\|A\|^2$.
4: Compute $X = A\bar{W}$ and set $Y = \bar{W}$.

Unfortunately, in finite precision arithmetic, Gram LRA suffers from instability due to the computation of the Gram matrix (whose condition number is $\kappa(A)^2$). Nevertheless, in **[9]** we perform an error analysis of this approach based on eigenvector perturbation theory which yields remarkably sharp error bounds. The following result shows that Gram LRA is actually not as unstable as one may think.

Theorem 6.4 (Theorem 3.1 of **[9]**). *Let $A \in \mathbb{R}^{m\times n}$ and let $\bar{A}$ be its rank-k truncated SVD. Let the approximate truncated SVD $A\bar{W}\bar{W}^T$ be computed with Algorithm 6.7 with precision parameters ε and u controlling the truncation error and the rounding errors, such that $\|G - W\Lambda W^T\| \leq c_{m,n}u\|A\|^2$ and $\|A - \bar{A}\| \leq \varepsilon\|A\|$. Then*

$$\|A - A\bar{W}\bar{W}^T\| \leq c_{m,n,k}\left(\varepsilon + \min\left(\kappa(\bar{A})u, \sqrt{u}\right)\right)\|A\| \tag{6.9}$$

where $c_{m,n,k}$ is a constant depending only on the dimensions m, n, and k, and $\kappa(\bar{A}) = \sigma_1/\sigma_k$ is the generalized condition number of $\bar{A}$.

The term $\min(\kappa(\bar{A})u, \sqrt{u})$ in (6.9) shows that not only does the error grow only linearly (and not quadratically) with the condition number $\kappa(\bar{A})$, but moreover it can never exceed $\sqrt{u}$, even for ill-conditioned matrices. Therefore, as long as we use a precision u that is *double* the target accuracy ε ($u = \varepsilon^2$), the effect of finite precision arithmetic should go unnoticed. For example, with double precision arithmetic ($u \approx 10^{-16}$), at least eight significant digits of accuracy are guaranteed, so that Algorithm 6.7 can handle truncation thresholds as small as $\varepsilon = 10^{-8}$. Since truncation thresholds in applications involving LRA are often larger, this algorithm can certainly be successful.

In **[9]**, we also propose two new ideas to improve Gram LRA in a finite precision setting. The first is to use mixed precision iterative refinement to refine the small eigenvalues of the Gram matrix, which can lead to a significant accuracy improvement in some cases. The second idea is to accelerate the final multiplication $A\bar{W}$ by performing it in lower precision, which can be done without affecting the accuracy because this operation is much less sensitive to rounding errors than the operations involving the Gram matrix.

6.7 Optimal quantization of low-rank matrices

Thinking about LRA as an optimization problem provides significant insight. As mentioned at the beginning of this chapter, the optimal LRA is given by the truncated SVD, that is, the solution of the optimization problem

$$\min_{X\in\mathbb{R}^{m\times k}, Y\in\mathbb{R}^{n\times k}} \|A - XY^T\| \tag{6.10}$$

is the rank-k truncated SVD.

However, this assumes exact arithmetic. In finite precision arithmetic, we wish to find an LRA while also storing its coefficients in a given floating-point arithmetic with t bits of significand. The optimization problem thus becomes

$$\min_{\widehat{X}\in\mathbb{F}_t^{m\times k},\widehat{Y}\in\mathbb{F}_t^{n\times k}} \|A-\widehat{X}\widehat{Y}^T\|, \tag{6.11}$$

where $\mathbb{F}_t$ is the set of floating-point numbers with t bits of significand. $\widehat{X}$ and $\widehat{Y}$ are commonly referred to as "quantized", because their coefficients can only take a finite, discrete number of possible values.

We can decouple the truncation and quantization errors by considering the problem

$$\min_{\widehat{X}\in\mathbb{F}_t^{m\times k},\widehat{Y}\in\mathbb{F}_t^{n\times k}} \|XY^T-\widehat{X}\widehat{Y}^T\|, \tag{6.12}$$

where XY^T is an exact (unquantized) rank-k matrix (which can, but need not, be the optimal LRA minimizing $\|A-XY^T\|$ in exact arithmetic). It would seem natural to think that the matrices $\widehat{X}=\mathrm{fl}(X)$ and $\widehat{Y}=\mathrm{fl}(Y)$ obtained by rounding the matrices X and Y would be optimal, since they certainly minimize the errors $\|X-\widehat{X}\|$ and $\|Y-\widehat{Y}\|$. Crucially, and surprisingly, they are actually far from optimal.

In **[42]**, we make a first step towards optimally quantizing low-rank matrices by considering the rank-one case ($k=1$). Hence, we focus on the problem

$$\min_{\widehat{x}\in\mathbb{F}_t^m,\widehat{y}\in\mathbb{F}_t^n} \|xy^T-\widehat{x}\widehat{y}^T\|. \tag{6.13}$$

The coefficients of the matrix $\widehat{x}\widehat{y}^T$ are quite special: they are the product of two t-bit floating-point numbers. We therefore take an interest in the set of such numbers, denoted $\mathbb{F}_t\mathbb{F}_t$. We show that this set is much denser than $\mathbb{F}_t$: the maximum relative distance between any real number and a number from $\mathbb{F}_t$ is the unit roundoff $u=2^{-t}$, but the same distance to a number from $\mathbb{F}_t\mathbb{F}_t$ is approximately $2^{-1.6t}$ (see Corollary 3.4 and Figure 3.2 of **[42]**). An analogy with multiword arithmetic can be made: real numbers can be approximated by the sum of two t-bit numbers with a relative accuracy of $2t$ bits; in **[42]** we show that the product of two t-bit numbers provides instead a relative accuracy of $1.6t$ bits.

We can exploit this observation as follows: given a real coefficient $(xy^T)_{ij}=x_iy_j$ of the matrix xy^T that we seek to quantize, we can approximate it optimally by selecting the quantized numbers $\widehat{x}_i,\widehat{y}_j\in\mathbb{F}_t$ such that $\widehat{x}_i\widehat{y}_j$ is the nearest number in $\mathbb{F}_t\mathbb{F}_t$ from x_iy_j. However, two obstacles must be overcome: first, we need to account for the constrained nature of the problem. The optimal $\widehat{x}_i$ that minimizes $|\widehat{x}_i\widehat{y}_j-x_iy_j|$ depends on which j is considered, but we can only choose one value. Second, we need to develop an algorithm to find the optimal $\widehat{x}_i\widehat{y}_j$ with tractable complexity.

The first step towards tackling these challenges is given by the following result, which characterizes the optimal quantized $\widehat{x}$ and $\widehat{y}$.

Theorem 6.5 (Theorem 4.5 of **[42]**). *Consider nonzero $x\in\mathbb{R}^m$, $y\in\mathbb{R}^n$, with $t\geq 1$. Problem (6.13) admits an optimum $\widehat{x},\widehat{y}$ such that*

$$\widehat{x}=\mathrm{fl}(\lambda x) \quad \textit{with } \lambda\in(1,2) \tag{6.14}$$

$$\widehat{y}=\mathrm{fl}(\mu y), \quad \textit{with } \mu=\begin{cases}\frac{x^T\widehat{x}}{\|\widehat{x}\|^2}, & \textit{if } \widehat{x}\neq 0\\ 0, & \textit{otherwise.}\end{cases} \tag{6.15}$$

Theorem 6.5 shows that the optimal $\widehat{x}, \widehat{y}$ are given by the quantization of suitably scaled factors λx and μy, where μ is determined by $\widehat{x}$ and thus by λ. This characterization has therefore reduced the optimization problem (6.13) in $m+n$ variables to an optimization problem in a single variable λ. Moreover, the search space for λ is reduced to $(1, 2)$, because the error is invariant under sign flip and multiplication by powers of two.

It now remains to develop a method to find the optimal $\lambda \in (1, 2)$. Algorithm 6.8 achieves this, by exploiting the key observation that there is only a finite, discrete number of values in the interval $(1, 2)$ that need to be tested: those which correspond to different values of $\mathrm{fl}(\lambda x)$. We provide in **[42]** (Lemma 5.1) an explicit formula for these "breakpoint" values, and we prove that there are at most $m2^t$. Therefore, Algorithm 6.8 is guaranteed to compute the optimal quantization in $O(mn2^t)$ complexity, simply by enumerating all the possible λ values and choosing the one that minimizes the quantization error. While this complexity is exponential in the number of bits t, it is polynomial in the matrix dimensions; it is therefore tractable for low precisions and large matrices.

Algorithm 6.8 Optimal rank-one quantization.

Input: $t \geq 1$ an integer, $x \in \mathbb{R}^m, y \in \mathbb{R}^n$.
Output: $\widehat{x}^* \in \mathbb{F}_t^m, \widehat{y}^* \in \mathbb{F}_t^n$ solutions to (6.13).

Initialize $\widehat{x}^* \leftarrow 0$, $\widehat{y}^* \leftarrow 0$
if $x = 0$ or $y = 0$ **then**
  exit
end if
$\mathbb{B} \leftarrow$ breakpoints(x) (see Lemma 5.1 in **[42]**)
Sort $\mathbb{B}$ in increasing order to obtain λ_j, $1 \leq j \leq J := \#\mathbb{B}$, $\lambda_0 \leftarrow 1$, $\lambda_{J+1} \leftarrow 2$.
for $j = 1$ **to** $J + 1$ **do**
  $\lambda \leftarrow (\lambda_{j-1} + \lambda_j)/2$
  $\widehat{x} \leftarrow \mathrm{round}(\lambda x)$
  $\mu \leftarrow x^T \widehat{x} / \|\widehat{x}\|^2$
  $\widehat{y} \leftarrow \mathrm{round}(\mu y)$
  if $\|xy^T - \widehat{x}\widehat{y}^T\| < \|xy^T - \widehat{x}^*(\widehat{y}^*)^T\|$ **then**
    $\widehat{x}^* \leftarrow \widehat{x}$, $\widehat{y}^* \leftarrow \widehat{y}$
  end if
end for

This optimal quantization method for rank-one matrices could be used as a building block for rank-k matrices, by quantizing their rank-one components one by one (see Research Problem 16.8). Moreover, it is directly useful in some applications where rank-one matrices naturally arise: this is notably the case of butterfly factorizations, see Section 13.3.

Chapter 7

Direct linear solvers

Given a linear system $Ax = b$, methods to compute the solution x have traditionally been divided into two broad classes:

- direct methods compute a factorization of matrix A to directly obtain the solution;
- iterative methods compute an approximate solution belonging to a lower dimensional subspace that is iteratively increased.

Broadly speaking, direct solvers are robust and easy to use, but have a high computational cost associated with the matrix factorization, whereas iterative solvers are more lightweight but their convergence is dependent on the construction of a "good" subspace via preconditioning. This chapter deals with direct solvers, and the next (Chapter 8) covers iterative ones. These two chapters set the stage for the approximate linear solvers that will be presented in subsequent chapters. In particular, Chapters 9 and 10 will describe approximate factorization methods (such as using block low-rank compression or low precision arithmetic) that can be viewed as either direct or iterative methods depending on the context. Indeed, these techniques can be used either to accelerate direct methods with a controlled loss of accuracy, or to build robust preconditioners that will ensure the fast convergence of iterative methods.

7.1 Dense systems

Given a square nonsingular matrix $A \in \mathbb{R}^{n\times n}$ and a right-hand side $b \in \mathbb{R}^n$, the standard Gaussian elimination method to solve the linear system $Ax = b$ consists of two phases:

1. compute the matrix factorization $PA = LU$, where L is a lower triangular matrix with diagonal elements equal to 1, U is an upper triangular matrix, and P is a permutation matrix;
2. solve the triangular systems $Ly = Pb$ and $Ux = y$ by substitution.

This approach directly yields the solution vector x and is thus referred to as a *direct* method.

The LU factorization can be replaced by an $\mathrm{LDL}^{\mathrm{T}}$ or LL^{T} one if the matrix is symmetric indefinite or symmetric positive definite, respectively. Throughout the rest of this manuscript we will focus on the unsymmetric case; unless otherwise mentioned, the presented ideas also apply to the symmetric case.

We first describe how to compute LU factorization for general dense matrices in this section. We will then discuss the multifrontal method for sparse matrices in Section 7.2.

Throughout this section, for the sake of the simplicity of the presentation, we discuss how to compute a decomposition $A = LU$ with no permutation matrix P. We will discuss how to integrate pivoting in the factorization, and why it is important to do so, in Section 7.3.

The n^2 equations $a_{ij} = \sum_{k=1}^{\min(i,j)} \ell_{ik} u_{kj}$ yield the Doolittle formula for computing the entries of the L and U factors:

for $k = 1\colon n$ **do**
  for $j = k\colon n$ **do**
    $u_{kj} = a_{kj} - \sum_{i=1}^{k-1} \ell_{ki} u_{ij}$
  end for
  for $i = k+1\colon n$ **do**
    $\ell_{ik} = \left(a_{ik} - \sum_{j=1}^{k-1} \ell_{ij} u_{jk} \right) / u_{kk}$
  end for
end for

In order to achieve high performance, this formula should be applied in a blockwise fashion by partitioning the matrix A into $p \times p$ blocks A_{ij}, as outlined in Algorithm 7.1. The algorithm consists of two main operations that are performed at each step $k = 1\colon p$. UPDATE updates the kth block-column and kth block-row of the matrix with respect to the already factored part. FACTOR computes the LU factors associated with this block-row and block-column. This algorithm requires $2n^3/3$ flops, and the $O(n^3)$ part of the flops are in the matrix–matrix products in the UPDATE operation. Note that the LU factorization can be performed in-place by overwriting A_{ij} by L_{ij} if $i > j$, U_{ij} if $i < j$, or $L_{ij} - I + U_{ij}$ if $i = j$; thus no additional storage to the n^2 entries of the matrix is needed.

Algorithm 7.1 Partitioned dense LU factorization (left-looking).

Input: a $p \times p$ block matrix A.
Output: its LU factors L and U.

1: **for** $k = 1$ **to** p **do**
2: UPDATE:
3: $A_{kk} \leftarrow A_{kk} - \sum_{j=1}^{k-1} L_{kj} U_{jk}$.
4: **for** $i = k+1$ **to** p **do**
5: $A_{ik} \leftarrow A_{ik} - \sum_{j=1}^{k-1} L_{ij} U_{jk}$ and $A_{ki} \leftarrow A_{ki} - \sum_{j=1}^{k-1} L_{kj} U_{ji}$.
6: **end for**
7: FACTOR:
8: Compute the LU factorization $L_{kk} U_{kk} = A_{kk}$.
9: **for** $i = k+1$ **to** p **do**
10: Solve $L_{ik} U_{kk} = A_{ik}$ for L_{ik} and $L_{kk} U_{ki} = A_{ki}$ for U_{ki}.
11: **end for**
12: **end for**

There is much flexibility in the order in which the block operations can be applied, because many of them are independent. Algorithm 7.1 is referred to as "left-looking" because at each step k, the update of a given block A_{ik} requires accessing all the factored blocks to the left L_{ij}, $j = 1\colon k-1$. Thus, the matrix product $L_{ij} U_{jk}$ is performed at the latest possible step k; instead, it could be performed earlier, as early as step j, just after L_{ij} and U_{jk} have been obtained. This alternative ordering of operations leads to Algorithm 7.2, which is referred to as "right-looking", because at each step k, we access all the blocks to the right, in the trailing submatrix. While the distinction between left- and right-looking LU factorizations is usually not critical, there are some contexts, especially relating to the use of approximations, in which they achieve very

different performance (see Section 9.3 or Section 10.1.2). Note however that they are numerically equivalent: they produce the same LU factors, not only in exact arithmetic, but even in floating-point arithmetic. This is because in both cases a given block A_{ik} is updated by $\sum_{j=1}^{k-1} L_{ij}U_{jk}$ and the sum is evaluated in the same order. However, since each update $L_{ij}U_{jk}$ is independent, many other orders are possible, which would not be equivalent due to the non-associativity of floating-point arithmetic. This is something to consider, for example, when using a task-based parallelization.

Algorithm 7.2 Partitioned dense LU factorization (right-looking).

Input: a $p \times p$ block matrix A.
Output: its LU factors L and U.

for $k = 1$ **to** p **do**
  FACTOR:
    Compute the LU factorization $L_{kk}U_{kk} = A_{kk}$.
    for $i = k+1$ **to** p **do**
      Solve $L_{ik}U_{kk} = A_{ik}$ for L_{ik} and $L_{kk}U_{ki} = A_{ki}$ for U_{ki}.
    end for
  UPDATE:
    for $i = k+1$ **to** p **do**
      for $j = k+1$ **to** p **do**
        $A_{ij} \leftarrow A_{ij} - L_{ik}U_{kj}$
      end for
    end for
end for

7.2 Sparse systems: the multifrontal method

The LU factorization of a sparse matrix can be computed by applying Doolittle's formula while skipping any computation with a zero element. The difficulty is that the LU factors, while sparse themselves, become denser than the original matrix: for example, the element $u_{kj} = a_{kj} - \sum_{i=1}^{k-1} \ell_{ki}u_{ij}$ can be nonzero even if $a_{kj} = 0$. This is known as the fill-in property, and u_{kj} is referred to as a filled entry. Moreover, this equation shows that a given entry u_{kj} will be filled if and only if there exists $i < k$ such that $\ell_{ki}u_{ij} \neq 0$, that is, such that ℓ_{ki} and u_{ij} are both nonzero entries.

As a result, the amount of fill-in strongly depends on the matrix ordering. For this reason, in addition to the numerical factorization and substitution phases, sparse direct solvers also require a preprocessing, commonly called the analysis phase, which aims at both reordering the matrix to minimize fill-in and predicting the sparsity structure of its LU factors via a symbolic factorization. The usual way to do so is to work on the graph of the adjacency matrix; indeed, the entry (k, j) will be filled if there exists a node $i < k$ connecting k and j. Note that (i, k) and (i, j) may be filled entries themselves, so that the graph needs to be updated by adding edges to track the fill-in; this is known as the elimination graph. We refer to [180] for a detailed description of the analysis phase, which we omit here because it is a non-numerical phase that is therefore not particularly amenable to approximate computing.

To describe how to exploit sparsity in LU factorization, we take an example illustrated in Figures 7.1 to 7.3. Figure 7.1 shows a 5×5 two-dimensional mesh with a five-point stencil, which corresponds to a sparse matrix of order 25. The unknowns have been numbered according

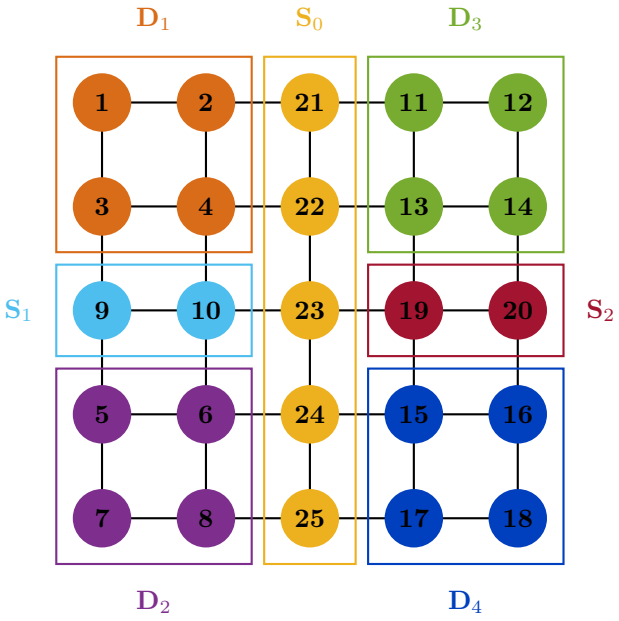


Figure 7.1: An example of a 2D mesh with five-point stencil. The mesh has been partitioned using nested dissection and the nodes of the mesh are labelled by the corresponding ordering.

to a nested dissection ordering [135]. This consists in partitioning the mesh in two equal parts using a vertex separator ($\mathbf{S}_0$), and recursively partitioning the two parts in the same way. We obtain a partitioning of the mesh into subdomains (we use the term subdomain to refer to both separators and the final domains $\mathbf{D}_1$, ..., $\mathbf{D}_4$). Then, the unknowns that belong to the same subdomain are numbered consecutively. This leads to the sparsity structure shown by the "$\times$" entries in Figure 7.2.

This nested dissection ordering strongly constraints the possible fill-in, as illustrated in Figure 7.2: the filled entries (marked with "f") are limited to within each subdomain and to the interaction between neighbor subdomains. Note that since the subdomain graphs are not complete, the entries within a given subdomain are not all filled. Some unknowns can be amalgamated, which leads to zero entries being treated as nonzero; this can increase the granularity and thus the efficiency of computations, at the price of additional fill-in. For the simplicity of illustration, in Figure 7.2 we have assumed that all the unknowns belonging to the same subdomain have all been amalgamated (entries marked with "o").

The nested dissection ordering moreover defines in a natural way the dependencies of the factorization: unknowns in independent subdomains can be eliminated concurrently. These dependencies can be organized under the form of a tree, as illustrated in Figure 7.3. Each node is associated with a subdomain, and hence a dense matrix of the form

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where A_{11} corresponds to the subdomain unknowns and A_{22} corresponds to the so-called contribution unknowns: those that are connected to the subdomain unknowns in the elimination graph and that will therefore receive a contribution. In this manuscript, we will focus on the multifrontal method [120, 118], which traverses this tree in a bottom-up order and performs two operations at each node. First, we assemble the dense matrix (called frontal matrix) by summing

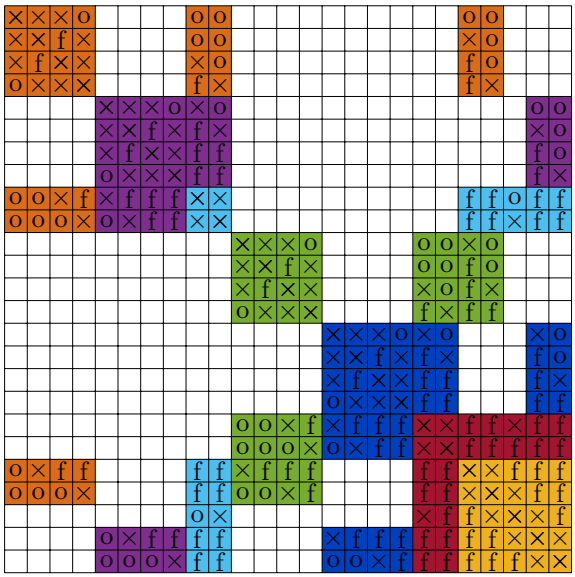


Figure 7.2: LU factors of the matrix associated with the mesh in Figure 7.1. White cells correspond to zero entries which are not stored, whereas colored entries are stored: "×" denotes a nonzero entry from the original matrix, "f" denotes a filled entry in the LU factors, and "o" denotes a zero entry which may be treated as nonzero if the subdomains are amalgamated.

the original entries of the matrix with the contribution entries from the child nodes. Second, we compute a partial LU factorization of the subdomain unknowns: $A_{11} = L_{11}U_{11}$, $L_{21} = A_{21}U_{11}^{-1}$, $U_{12} = L_{11}^{-1}A_{12}$ (fully colored cells in Figure 7.3) yields the part of the LU factors associated with this node, and the Schur complement $A_{22} \leftarrow A_{22} - L_{21}U_{12}$ (hatch lines cells) yields the contribution that will be stored until it is used to assemble the parent node.

The complexity of the multifrontal method can therefore be computed as

$$\sum_{\text{level } \ell} \#\text{nodes}(\ell) \times \mathcal{C}_{\text{dense}}\big(\text{nodesize}(\ell)\big),$$

where $\mathcal{C}_{\text{dense}}(m)$ is the complexity of processing a dense matrix of size $m \times m$. Consider a regular problem discretized on a d-dimensional domain of size $n = N^d$, where d is usually 2 or 3. In order to simplify the complexity formula, we can merge the levels by groups of size d in order to obtain "hypercross" separators; the root separator is of size $O(N^{d-1})$ and at each level it partitions the domain into 2^d subdomains, which will be further partitioned by separators 2^{d-1} times smaller. This therefore yields the formula:

$$\sum_{\ell=0}^{\log_2 N} 2^{d\ell} \times \mathcal{C}_{\text{dense}}\Big(\big(\frac{N}{2^\ell}\big)^{d-1}\Big). \tag{7.1}$$

By setting $d = 2$ or $d = 3$, we obtain the classical complexity bounds [135] reported in the first row of Table 9.1 (found in Chapter 9). In particular, the complexities for 3D problems, $O(n^{4/3})$ for storage and $O(n^2)$ for flops, are highly superlinear and illustrate the high computational cost of direct methods.

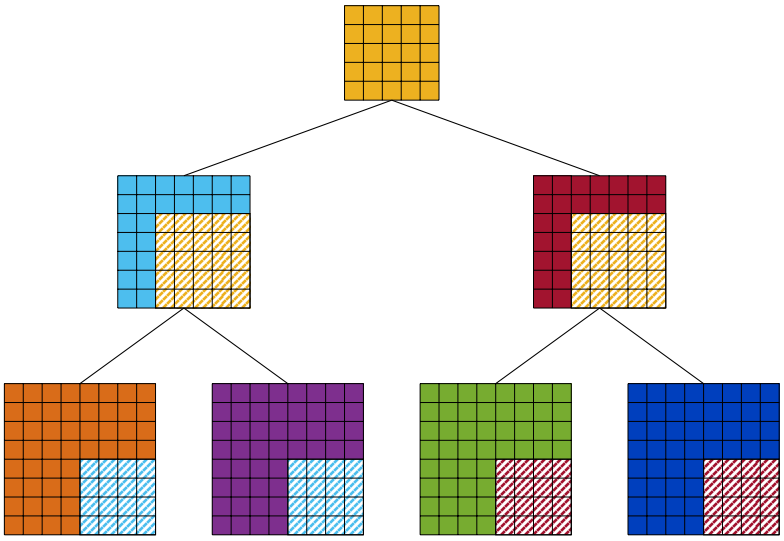


Figure 7.3: Multifrontal tree associated with the LU factors in Figure 7.2. The LU factors are computed following a bottom-up traversal of the tree, where at each node we compute a partial dense LU factorization (fully colored cells), as well as its Schur complement called the contribution block (hatch lines cells), which is summed to the parent node.

7.3 Stability and pivoting

We have the following standard result on the stability of LU factorization, which is a direct consequence of the fact that Doolittle's formula for computing the entries of the LU factors essentially takes the form of an inner product.

Theorem 7.1 (Theorem 9.3 in [162]). *If the LU factorization of $A \in \mathbb{R}^{n\times n}$ runs to completion then the computed LU factors $\widehat{L}$ and $\widehat{U}$ satisfy*

$$\widehat{L}\widehat{U} = A + \Delta A, \quad |\Delta A| \leq \gamma_n |\widehat{L}||\widehat{U}|. \tag{7.2}$$

We also have the following result on the stability of substitution.

Theorem 7.2 (Theorem 8.5 in [162]). *Let the triangular system $Tx = b$, where $T \in \mathbb{R}^{n\times n}$ is nonsingular, be solved by substitution, with any ordering. Then the computed $\widehat{x}$ satisfies*

$$(T + \Delta T)\widehat{x} = b, \quad |\Delta T| \leq \gamma_n |T|. \tag{7.3}$$

Combining Theorem 7.1 and Theorem 7.2 yields the following result for the solution of linear systems.

Theorem 7.3 (Theorem 9.4 in [162]). *Let $A \in \mathbb{R}^{n\times n}$ and suppose that its LU factorization produces computed LU factors $\widehat{L}$ and $\widehat{U}$. Let the system $\widehat{L}\widehat{U}x = b$ be solved by substitution. Then the computed solution $\widehat{x}$ satisfies*

$$(A + \Delta A)\widehat{x} = b, \quad |\Delta A| \leq \gamma_{3n} |\widehat{L}||\widehat{U}|. \tag{7.4}$$

All the above results are componentwise backward error bounds, which directly imply corresponding normwise bounds proportional to $\gamma_n \||\widehat{L}||\widehat{U}|\|$. While these bounds are thus proportional to the unit roundoff, they are not sufficient to conclude on the stability of direct methods. This is

because we would like the error to be bounded in terms of $\|A\|$, rather than in terms of $\||\widehat{L}||\widehat{U}|\|$. The ratio between these two quantities is traditionally expressed in terms of the *growth factor* ρ_n, which is defined as the ratio between the largest element appearing during the LU factorization of A and the largest element of A. Unfortunately, ρ_n can be arbitrarily large. This is because of the division by the diagonal element u_{kk} in Doolittle's formula, which can be arbitrarily small. In fact, if $u_{kk} = 0$, the algorithm breaks down.

This motivates the use of pivoting strategies, which at each step k select a diagonal pivot u_{kk} that is sufficiently large. Several strategies can be used, with different tradeoffs between stability and cost. The most widely used and popular strategy is *partial pivoting*, which at each step k selects the largest element a_{jk} in column k and swaps rows j and k. LU factorization with partial pivoting therefore computes the decomposition $PA = LU$, where P is a row permutation matrix.

For partial pivoting, we have the bound $\||\widehat{L}||\widehat{U}|\|_\infty \leq n^2 \rho_n \|A\|_\infty$, which yields the following result.

Theorem 7.4 (Theorem 9.5 in [162]). *Let $A \in \mathbb{R}^{n\times n}$ and suppose that Gaussian elimination with partial pivoting produces a computed solution $\widehat{x}$ to $Ax = b$. Then*

$$(A + \Delta A)\widehat{x} = b, \quad \|\Delta A\|_\infty \leq n^2 \gamma_{3n} \rho_n \|A\|_\infty \tag{7.5}$$

The stability of LU factorization with partial pivoting is therefore conditioned on a small growth factor ρ_n. Unfortunately, the best general bound for ρ_n is 2^{n-1}, which can be attained for some very special matrices. However, in practice, this bound is very pessimistic and ρ_n is almost always a small constant. Therefore, LU factorization with partial pivoting is generally considered to be stable in a practical sense. We refer to [162, Chapter 9] for a more in-depth discussion of the growth factor and pivoting strategies for dense systems.

We conclude by briefly discussing pivoting strategies for sparse systems. The difficulty with pivoting for sparse matrices is that, as explained in the previous section, the amount of fill-in depends on the matrix ordering; therefore, by pivoting, that is, changing this ordering, we create additional fill-in, and moreover we complexify the algorithm by making the sparsity structure of the LU factors unpredictable. Nevertheless, pivoting remains essential for the stability of sparse direct solvers, which has motivated the development of pivoting strategies that seek to reduce the number of permutations while maintaining some stability. This is notably the case of threshold partial pivoting [118, Chap. 7], which avoids swapping out the diagonal element a_{kk} if its magnitude is within a given factor τ of the largest element in the column. Another more aggressive strategy is static pivoting [186], which replaces diagonal elements a_{kk} which are smaller than τ by τ, hence introducing a perturbation to the system. Several sparse direct solvers incorporate static pivoting strategies as an option, such as MUMPS [60], **[13]**, which implements the approach proposed by Duff and Pralet [119], or as the default, such as SuperLU_DIST [185] and PARDISO [233].

Chapter 8

Iterative linear solvers

Iterative methods for solving $Ax = b$ build a sequence of iterates $x_1, x_2, \ldots$, hopefully converging towards a good approximation of the solution x. Many different iterative methods can be considered, depending on the type of matrix (general, symmetric, positive definite, etc.) and on the desired tradeoff between robustness and cost. In this chapter, and in most of this manuscript, we will focus on the generalized minimum residual (GMRES) [231] method. We refer to [232] for an extensive discussion of both GMRES and other iterative methods.

The contributions described in this chapter are based on the following papers: **[11]**, **[12]** for Section 8.2; **[40]** for Section 8.3.1; **[47]** for Section 8.4.

8.1 GMRES

The basic idea at the foundation of most iterative methods is to seek an approximate solution belonging to a subspace $\mathcal{K}_m$ of lower dimension $m \ll n$. In particular, Krylov methods build the Krylov subspace

$$\mathcal{K}_m = \operatorname{span}\left\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\right\}, \tag{8.1}$$

where $r_0 = b - Ax_0$ and x_0 is an initial guess.

An orthonormal basis V_m of this subspace can be built progressively by repeated matrix–vector multiplication with A. This is known as Arnoldi's iteration; the variant outlined below uses modified Gram–Schmidt (MGS) orthonormalization.

$\beta = \|r_0\|_2$
$v_1 = r_0/\beta$
for $j = 1 : m$ **do**
  $w_j = Av_j$
  for $i = 1 : j$ **do**
    $h_{ij} = w_j^T v_i$
    $w_j = w_j - h_{ij} v_i$
  end for
  $h_{j+1,j} = \|w_j\|_2$. If $h_{j+1,j} = 0$ stop.
  $v_{j+1} = w_j / h_{j+1,j}$
end for

Assuming this process does not stop early, it yields an $n \times (m+1)$ matrix V_{m+1} with orthonormal columns and an $(m+1) \times m$ Hessenberg matrix $\bar{H}_m$, satisfying the relation

$$AV_m = V_{m+1} \bar{H}_m. \tag{8.2}$$

Consider now an approximate solution x_m belonging to the affine subspace $x_0 + \mathcal{K}_m$, that is, that can be written under the form $x_m = x_0 + V_m y_m$ with $y_m \in \mathbb{R}^m$. Relation (8.2) then yields

$$\begin{aligned} b - Ax_m &= b - A(x_0 + V_m y_m) \\ &= r_0 - AV_m y_m \\ &= \beta v_1 - V_{m+1}\bar{H}_m y_m \\ &= V_{m+1}(\beta e_1 - \bar{H}_m y_m), \end{aligned} \tag{8.3}$$

where $e_1 = (1, 0, \ldots, 0) \in \mathbb{R}^{m+1}$. Since the columns of V_{m+1} are orthonormal, (8.3) yields

$$\|b - Ax_m\|_2 = \|\beta e_1 - \bar{H}_m y_m\|_2. \tag{8.4}$$

This shows that the approximant x_m that minimizes the norm of the residual $b - Ax_m$ is given by $x_m = x_0 + V_m y_m$, where y_m is the solution of the $(m+1) \times m$ least-squares problem $\min \|\beta e_1 - \bar{H}_m y_m\|_2$, which is inexpensive to solve when m is small. This key observation leads to the GMRES method, outlined in Algorithm 8.1.

Algorithm 8.1 GMRES.

Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, initial guess x_0, number of iterations m, right preconditioner M^{-1}.
Output: an approximate solution $x_m \in \mathbb{R}^n$ to $Ax = b$.

```
r_0 = b − Ax_0
β = ‖r_0‖_2
v_1 = r_0/β
for j = 1: m do
    w_j = AM^{-1} v_j
    for i = 1: j do
        h_ij = w_j^T v_i
        w_j = w_j − h_ij v_i
    end for
    h_{j+1,j} = ‖w_j‖_2. If h_{j+1,j} = 0 stop.
    v_{j+1} = w_j/h_{j+1,j}
end for
Compute y_m the minimizer of ‖βe_1 − H̄_m y‖_2.
x_m = x_0 + M^{-1} V_m y_m
```

In practice, the least squares problem is solved by transforming the Hessenberg matrix to upper triangular form with plane rotations. This can be done progressively at each iteration, and thanks to (8.4), this provides as a byproduct a criterion to stop GMRES early if the residual is sufficiently small. See [232, section 6.5.3] for details.

Since the Krylov subspaces $\mathcal{K}_m$ at each iteration m are nested, the residual is non-increasing in exact arithmetic. Moreover, at $m = n$, the subspace spans the entire $\mathbb{R}^n$ which contains the solution, so GMRES eventually converges. The hope is that in practice, a small number of iterations $m \ll n$ will suffice to achieve a satisfactory approximation. Unfortunately, in general, there is no guarantee on the convergence rate of GMRES. A result from Greenbaum et al. [145] states that for any nonincreasing convergence curve and any eigenvalue distribution, one can find a matrix with these eigenvalues for which GMRES follows this curve. The convergence of GMRES does depend on the matrix, however, which motivates the idea of preconditioning.

Preconditioning consists in replacing the matrix A by a preconditioned matrix $\widetilde{A}$, such as $\widetilde{A} = M^{-1}A$ (left preconditioning) or $\widetilde{A} = AM^{-1}$ (right preconditioning, as in Algorithm 8.1).

The matrix M is the preconditioner: ideally, its inverse M^{-1} should be cheap to compute and to apply, yet a good approximation of A^{-1}. The term "preconditioning" refers to the fact that, if $M^{-1} \approx A^{-1}$, the preconditioned matrix will have a much smaller condition number than A. When A is symmetric positive definite, the condition number $\kappa(\widetilde{A})$ is indeed a key quantity that controls the convergence rate of GMRES (and other iterative methods). However, for general matrices, it is important to note that the convergence of GMRES is not determined by $\kappa(\widetilde{A})$. Still, a good approximation $M^{-1} \approx A^{-1}$, and thus a small $\kappa(\widetilde{A})$, will often lead to a faster convergence.

A variant of right-preconditioned GMRES of particular interest in the context of approximate computing is flexible GMRES (FGMRES) [230]. FGMRES makes two slight modifications to Algorithm 8.1: when computing $w_j = AM^{-1}v_j$, we store the intermediate result $z_j = M^{-1}v_j$, and we replace $x_m = x_0 + M^{-1}V_m y_m$ with $x_m = x_0 + Z_m y_m$ by reusing the stored basis $Z_m = [z_1, \ldots, z_m]$. FGMRES thus requires storing an extra basis Z_m, but allows for the preconditioner M^{-1} to be variable, that is, to change from one iteration to the other. This is notably useful when the action of M^{-1} is obtained by an iterative solver itself.

The construction of the basis V_m requires $O(nm)$ storage and $O(nm^2)$ flops, so that the cost of GMRES can become prohibitive if a large number of iterations m is needed. To reduce this cost, the method can be restarted after a fixed number m of iterations have been performed, as described in Algorithm 8.2. This may slow down the convergence of the method but keeps its cost per iteration bounded.

Algorithm 8.2 Restarted GMRES.

Input: $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$, initial guess x_0, restart size m, right preconditioner M^{-1}.
Output: An approximate solution $x \in \mathbb{R}^n$ to $Ax = b$.

```
x = x_0
while not converged do
  r = b − Ax
  β = ‖r‖_2
  v_1 = r/β
  for j = 1: m do
    w_j = AM^{-1}v_j
    for i = 1: j do
      h_ij = w_j^T v_i
      w_j = w_j − h_ij v_i
    end for
    h_{j+1,j} = ‖w_j‖_2. If h_{j+1,j} = 0 stop.
    v_{j+1} = w_j/h_{j+1,j}
  end for
  Compute y_m the minimizer of ‖βe_1 − H̄_m y‖_2.
  x = x + M^{-1}V_m y_m
end while
```

8.2 Stability of GMRES

As mentioned previously, in exact arithmetic, GMRES will eventually converge to the solution x. In finite precision arithmetic, this is no longer true, so the question is: what accuracy can be attained by GMRES, and under which conditions?

8.2.1 Unpreconditioned GMRES

The main difficulty to study the stability of GMRES is that in finite precision arithmetic the Krylov basis V_m no longer has exactly orthonormal columns. GMRES was first proven backward stable in 1995 by Drkošová et al. [116] assuming the use of Householder orthonormalization, which produces a basis V_m with a loss of orthonormality of order the unit roundoff, regardless of the ill-conditioning of the matrix A. However, in practice, MGS orthonormalization is preferred and more widely used. In this case, the basis V_m can be affected by a significant loss of orthonormality. Despite this, in 2006 Paige et al. [219] proved that GMRES with MGS orthonormalization is backward stable, with the following result.

Theorem 8.1 (section 8 of [219]). *Let $Ax = b$ be solved with GMRES with MGS orthonormalization (Algorithm 8.1) and no preconditioner ($M = I$) in a precision with unit roundoff u. Provided that A is not numerically singular, that is, $\kappa(A)u \ll 1$, then there exists $k \leq n$ such that the iterate $\widehat{x}_k$ computed at iteration k satisfies the backward and forward error bounds*

$$\frac{\|b - A\widehat{x}_k\|}{\|A\|\|\widehat{x}_k\| + \|b\|} \leq c(n,k)u \tag{8.5}$$

and

$$\frac{\|x - \widehat{x}_k\|}{\|x\|} \leq c(n,k)\kappa(A)u, \tag{8.6}$$

where $c(n,k)$ is a low degree polynomial in n and k.

The analysis of Paige et al. is based on the key observation that V_m remains well-conditioned until it has fully lost its orthonormality, at which point the solution x must lie in the range of the Krylov subspace, that is, GMRES must have converged. The specific iteration k at which this happens is unknown, and in view of the previous discussion on the convergence of GMRES in exact arithmetic, it could be as large as $k = n$. Therefore, Theorem 8.1, and other similar stability results, cannot say anything about the convergence rate of GMRES in finite precision; it only bounds the attainable backward and forward errors.

8.2.2 Preconditioned GMRES

Note that Theorem 8.1 only applies to unpreconditioned GMRES. Indeed, the analysis of Paige et al. assumes matrix–vector products in standard floating-point arithmetic, but if a preconditioner M^{-1} is used, then the products with the preconditioned matrix $\widetilde{A}$ introduce additional rounding errors and the possibility of cancellation. Nevertheless, the analysis can be generalized to account for these additional errors. We do so in **[11]** and obtain the following result.

Theorem 8.2 (Theorem 3.1 of **[11]**). *Let $\widetilde{A}x = \widetilde{b}$ be solved with GMRES with MGS orthonormalization (Algorithm 8.1), carrying out its operations in a precision with unit roundoff u_g, except for the products with $\widetilde{A}$, which satisfy instead*

$$\mathrm{fl}(\widetilde{A}v) = \widetilde{A}v + f, \quad \|f\| \leq \varepsilon_p\|\widetilde{A}\|\|v\|, \tag{8.7}$$

Provided that $\kappa(\widetilde{A})(\varepsilon_p + u_g) \ll 1$, then there exists $k \leq n$ such that the iterate $\widehat{x}_k$ computed at iteration k satisfies the backward and forward error bounds

$$\frac{\|\widetilde{b} - \widetilde{A}\widehat{x}_k\|}{\|\widetilde{A}\|\|\widehat{x}_k\| + \|\widetilde{b}\|} \leq c(n,k)(\varepsilon_p + u_g) \tag{8.8}$$

and

$$\frac{\|x-\widehat{x}_k\|}{\|x\|} \le c(n,k)\kappa(\widetilde{A})(\varepsilon_p + u_g), \tag{8.9}$$

where $c(n,k)$ is a low degree polynomial in n and k.

Theorem 8.2 uses a general model of error (8.7) that allows for arbitrary products with $\widetilde{A}$, whose stability is quantified by a parameter ε_p, whereas the rest of the GMRES operations are carried out in standard floating-point arithmetic with unit roundoff u_g. This model can cover a wide range of different preconditioners, and applies to both left and right preconditioning. In particular, if the preconditioner comes from an approximate LU factorization ($M^{-1} = U^{-1}L^{-1}$), and is applied by substitution in precision u_p, standard error analysis shows that the model (8.7) holds with $\varepsilon_p \approx n^2\kappa(A)u_p$. Thus, (8.8) yields a backward error bound of order $\kappa(A)u_p + u_g$, which suggests that preconditioned GMRES is not necessarily stable due to the appearance of a $\kappa(A)$ term in the bound. This motivates the idea of applying the preconditioner in a precision higher than the rest of the operations ($u_p \ll u_g$), as suggested by Carson and Higham [87] in the context of GMRES-based iterative refinement (see section 10.2).

Note that, when left preconditioning is in play, we must distinguish the preconditioned system $\widetilde{A}x = \widetilde{b}$ from the original system $Ax = b$. In particular, the backward error bound (8.8) applies to the preconditioned system. A bound for the original system can be obtained as follows. Let M_L^{-1} be the left preconditioner, so that $\widetilde{A} = M_L^{-1}A$ and $\widetilde{b} = M_L^{-1}b$, and assume that $\|\widetilde{A}\widehat{x}_k - \widetilde{b}\| \le \varepsilon(\|\widetilde{A}\|\|\widehat{x}_k\| + \|\widetilde{b}\|)$ for some $\varepsilon \ge 0$. Then the inequality

$$\|A\widehat{x}_k - b\| = \|M_L(\widetilde{A}\widehat{x}_k - \widetilde{b})\| \le \|M_L\|\varepsilon(\|\widetilde{A}\|\|\widehat{x}_k\| + \|\widetilde{b}\|) \le \kappa(M_L)\varepsilon(\|A\|\|\widehat{x}_k\| + \|b\|)$$

shows that an extra $\kappa(M_L)$ term appears. This manipulation to transform a backward error bound on the preconditioned system to one on the original system usually produces pessimistic bounds. For this reason, in practice, we rather focus on the forward error bounds such as (8.9).

8.2.3 Flexible GMRES

The stability of flexible GMRES (FGMRES) has also been studied. Backward error analyses of FGMRES with MGS orthonormalization were carried out by Arioli et al. in [69, 68]. In particular, [68, Theorem 3.1] computes a backward error bound for general preconditioners. We state here a slightly modified version corresponding to Theorem 5.3 of **[12]** (see **[12]** for a discussion of the differences with [68, Theorem 3.1]).

Theorem 8.3 (Theorem 5.3 of **[12]**). *Let $Ax = b$ be solved with FGMRES with MGS orthonormalization in a precision with unit roundoff u. Let $k \le n$ the iteration at which b lies numerically in the range of AZ_k, in the sense that for all $\phi > 0$ we have*

$$\sigma_{\min}([b\phi, AZ_k]) \le c(n,k)u\frac{\|A\|\|Z_k\|}{\|AZ_k\|}\|[b\phi, AZ_k]\| \qquad \text{and} \qquad \sigma_{\min}(AZ_k) \gg u\|A\|\|Z_k\|.$$

If at this iteration k, the condition $\kappa(Z_k)u \ll 1$ holds, then the computed iterate $\widehat{x}_k$ satisfies the backward and forward error bounds

$$\frac{\|b - A\widehat{x}_k\|}{\|A\|\|\widehat{x}_k\| + \|b\|} \le c(n,k)\kappa(Z_k)u \tag{8.10}$$

and

$$\frac{\|x-\widehat{x}_k\|}{\|x\|} \le c(n,k)\kappa(Z_k)\kappa(A)u, \tag{8.11}$$

where $c(n,k)$ is a low degree polynomial in n and k.

Theorem 8.3 proves that as long as the basis Z_k does not become numerically singular before convergence, FGMRES will converge to a backward error of order $u\kappa(Z_k)$. With no preconditioner, $Z_k = V_k$ is a matrix with orthonormal columns, so $\kappa(Z_k)$ is a constant and we recover the stability of unpreconditioned GMRES. However, the term $\kappa(Z_k)$ shows that, just like left-preconditioned GMRES, FGMRES may also not be backward stable for general preconditioners. Importantly, Arioli et al. [68] show that if the preconditioner is specialized to LU factors $\widehat{L}\widehat{U}$ computed in a precision $\sqrt{u} \le u_f \le u$, if the solution is initialized to $x_0 = \widehat{U}^{-1}\widehat{L}^{-1}b$, and if $\kappa(A)u_f \ll 1$, then the bound (8.10) can be sharpened to become of order u.

8.2.4 Modular framework for the error analysis of GMRES

As the discussion in the previous sections illustrates, the error analysis of GMRES and its variants is a complex question. Unfortunately, the previously mentioned analyses only cover a tiny subset of common GMRES variants. Indeed, there is a wide range of possibilities, between the choice of preconditioner, preconditioning style (left, right, flexible, split), orthonormalization method (Householder, classical or modified Gram-Schmidt, etc.), restart criterion, and other approximation techniques (mixed precision, randomization [73], basis compression [59], block orthonormalization and communication avoiding approaches [91, 165], etc.). Moreover, because these previous analyses are long, sophisticated, and were not made to be modular, extending one of them to derive a new error analysis for a given variant of GMRES is generally far from straightforward.

Motivated by this issue, in **[12]**, we set out to develop a modular framework that simplifies the process of deriving error bounds for as many GMRES variants as possible. This framework also allows for unifying and comparing the existing analyses.

Model 8.1 (Modular framework for GMRES; sections 3.1 and 3.2 of **[12]**). *We model a GMRES method for solving $Ax = b$ as the following process:*

1. *Compute the preconditioned matrix product $C_k = M_L^{-1}AZ_k$ such that*

$$\widehat{C}_k = M_L^{-1}AZ_k + \Delta C_k, \quad \|\Delta C_k\| \le \varepsilon_c \|M_L^{-1}AZ_k\|, \tag{8.12}$$

   where M_L is the left preconditioner and Z_k is the basis.

2. *Compute the preconditioned right-hand side $\widetilde{b} = M_L^{-1}b$ such that*

$$\widehat{b} = \widetilde{b} + \Delta b, \quad \|\Delta b\| \le \varepsilon_b \|\widetilde{b}\|. \tag{8.13}$$

3. *Solve the least-squares problem $y_k = \arg\min_y \|\widetilde{b} - C_k y\|$ such that*

$$\begin{aligned} \widehat{y}_k &= \arg\min_y \|\widehat{b} + \Delta_{\text{ls}}^b - (\widehat{C}_k + \Delta_{\text{ls}}^c)y\|, \\ \|\left[\Delta_{\text{ls}}^b, \Delta_{\text{ls}}^c\right] e_j\| &\le \varepsilon_{\text{ls}} \|\left[\widehat{b}, \widehat{C}_k\right] e_j\|, \quad j = 1\colon k+1. \end{aligned} \tag{8.14}$$

4. *Compute the solution $x_k = Z_k y_k$ such that*

$$\widehat{x}_k = Z_k \widehat{y}_k + \Delta x_k, \quad \|\Delta x_k\| \le \varepsilon_x \|Z_k\|_F \|\widehat{y}_k\|. \tag{8.15}$$

   In addition, we also make the following assumptions.

5. *We assume all accuracy parameters to be sufficiently less than 1: $0 \le \varepsilon_c, \varepsilon_b, \varepsilon_{\text{ls}}, \varepsilon_x \ll 1$.*

6. *We assume Z_k not to be numerically singular to accuracy ε_x: $\kappa(Z_k)\varepsilon_x \ll 1$.*

7. *We assume that there exists a "key" iteration $k \leq n$ at which $\widetilde{b}$ lies in the range of $M_L^{-1}AZ_k$ and $M_L^{-1}AZ_k$ is not numerically singular, that is, for all $\phi > 0$,*

$$\begin{aligned} \sigma_{\min}\big([\widetilde{b}\phi, M_L^{-1}AZ_k]\big) &\ll (\varepsilon_c + \varepsilon_b + \varepsilon_{\text{ls}})\|[\widetilde{b}\phi, M_L^{-1}AZ_k]\| \\ \sigma_{\min}\big(M_L^{-1}AZ_k\big) &\gg (\varepsilon_c + \varepsilon_b + \varepsilon_{\text{ls}})\|M_L^{-1}AZ_k\|. \end{aligned} \tag{8.16}$$

Items 1–4 of Model 8.1 are parameterized error models for the core linear algebra kernels used by GMRES; specializing a given GMRES implementation to this model simply requires analyzing these kernels independently (hence the modularity of the framework). Items 5–6 are mostly innocuous assumptions. Finally, item 7 is the key assumption that is the most complex to check: it essentially assumes that GMRES will not fail before computing a basis whose span contains the solution. This assumption is tightly linked to the orthonormalization method that is considered. For example, we have already explained that it is satisfied for both Householder and MGS orthonormalization.

Under Model 8.1, we prove the following result.

Theorem 8.4 (Theorem 3.1 in **[12]**). *If a GMRES method satisfying Model 8.1 is used to solve $Ax = b$, then at the "key" iteration $k \leq n$, the computed iterate $\widehat{x}_k$ satisfies the backward and forward error bounds*

$$\frac{\|M_L^{-1}b - M_L^{-1}A\widehat{x}_k\|}{\|M_L^{-1}A\|\|\widehat{x}_k\| + \|M_L^{-1}b\|} \lesssim c(n,k)\xi, \tag{8.17}$$

and

$$\frac{\|\widehat{x}_k - x\|}{\|x\|} \lesssim c(n,k)\kappa(M_L^{-1}A)\xi, \tag{8.18}$$

where

$$\xi = \alpha\varepsilon_c + \beta\varepsilon_b + \beta\varepsilon_{ls} + \lambda\varepsilon_x \tag{8.19}$$

with

$$\alpha = \frac{\|M_L^{-1}AZ_k\|}{\|M_L^{-1}A\|\sigma_{\min}(Z_k)}, \quad \beta = \max(1,\alpha), \quad \lambda = \kappa(Z_k), \tag{8.20}$$

and with $c(n,k)$ a low degree polynomial in n and k.

Our framework, with Theorem 8.4, can be used to derive error bounds for many GMRES variants. First of all, we can check that we recover the same bounds, under the same conditions, than all the previously stated results. For example, with no preconditioner, $Z_k = V_k$ has orthonormal columns and so α, β, and λ are all constants; moreover $\varepsilon_b = 0$. If all steps are performed in the same precision with unit roundoff u, then standard error analysis readily yields $\varepsilon_c \equiv \varepsilon_{\text{ls}} \equiv \varepsilon_x \equiv c(n,k)u$. This, together with the proof from Paige et al. [219] of the existence of a key iteration satisfying (8.16) for MGS orthonormalization, recovers the backward stability of unpreconditioned GMRES proven in Theorem 8.1. Similarly, the results of Theorems 8.2 and 8.3 can also be recovered. Thus, Model 8.1 unifies existing analyses of GMRES under the same framework.

Even more interestingly, our framework can be used to obtain new results for variants of GMRES previously not covered by theoretical analyses. At the date of this writing, there have been four major new results that were obtained thanks to this framework. The first, which we provide in the same paper **[12]**, is the proof of the backward stability of GMRES with CGS2 orthonormalization (that is, classical Gram–Schmidt performed twice [140]). GMRES with CGS2

orthonormalization was conjectured to be stable for decades, and in fact a proof was given in 1995 by Drkošová et al. [116] under the assumption that the loss of orthogonality incurred with CGS2 is of order u. This assumption was later proved in 2005 by Giraud et al. [140], but under another assumption, namely that the matrix that is orthonormalized, $[b\ \widehat{C}_k]$, is numerically full-rank [140, Theorem 2]. Unfortunately, this assumption cannot be satisfied for all $k \leq n$ in our context since, if GMRES converges, b will eventually belong to the range of $\widehat{C}_k$ and cause the matrix $[b\ \widehat{C}_k]$ to become numerically singular. A formal proof of stability requires to show that the least squares problem equivalence (8.4) remains valid even if the basis V_{m+1} in (8.3) has fully lost orthonormality; we do so in **[12]** and obtain the following result.

Theorem 8.5 (Theorem 6.2 of **[12]**). *Let $Ax = b$ be solved with GMRES with CGS2 orthonormalization in a precision with unit roundoff u. Provided that A is not numerically singular, that is, $\kappa(A)u \ll 1$, then there exists $k \leq n$ such that the iterate $\widehat{x}_k$ computed at iteration k satisfies the backward and forward error bounds*

$$\frac{\|b - A\widehat{x}_k\|}{\|A\|\|\widehat{x}_k\| + \|b\|} \leq c(n,k)u \tag{8.21}$$

and

$$\frac{\|x - \widehat{x}_k\|}{\|x\|} \leq c(n,k)\kappa(A)u, \tag{8.22}$$

where $c(n,k)$ is a low degree polynomial in n and k.

The second major application of our framework is the error analysis of s-step GMRES performed by Carson and Ma [92]. The s-step (also called communication-avoiding) GMRES method [165] consists in adding to the basis and orthonormalizing s vectors at a time. This can significantly reduce the communication and synchronization costs of the solver [165, sect. 3.6], [255]. However, in finite precision arithmetic, s-step GMRES has been observed to be less stable and this was conjectured to be due to the Krylov basis becoming more and more ill-conditioned as s increases [165]. Carson and Ma formally prove this fact by obtaining a backward error bound of order u times the condition number of the basis [92, Lemma 4].

The third major application of the framework is the error analysis of sketched GMRES [207] performed by Burke et al. [81]. Sketched GMRES consists in replacing the least-squares problem $\min \|M_L^{-1}b - M_L^{-1}AZ_k y\|_2$ by the sketched problem $\min \|SM_L^{-1}b - SM_L^{-1}AZ_k y\|_2$, where S is a suitably chosen random matrix with a small number of rows. This can reduce the cost of the orthonormalization because the preconditioned basis $Z_k = M_R^{-1}V_k$ can be constructed using a Krylov basis V_k that is not orthonormal, or rather partially orthonormalized with truncated Arnoldi approaches that only orthonormalize a selected subset of vectors [148]. The sketched least-squares problem is then solved via a QR factorization of the sketch $SM_L^{-1}AZ_k$, which is of small dimension. However, if V_k is not orthonormal, then the condition number $\kappa(Z_k)$ can grow rapidly with the number of iterations and make the method fail. This is formally proven by the analysis of Burke et al. [81] that relates the stability of sketched GMRES to $\kappa(Z_k)$. Burke et al. moreover show that the method can stabilized by restarting GMRES whenever $\kappa(Z_k)$ becomes too large, which is a form of iterative refinement (Chapter 10).

Finally, the fourth major application of the framework is to mixed precision preconditioned GMRES. Indeed, the modular error models in items 1–4 of Model 8.1 allow for using different precisions for different operations. In the following Section 8.3.1, we discuss the error bounds resulting from Theorem 8.4 and the mixed precision strategies that they reveal.

8.3 Mixed precision GMRES

GMRES offers many opportunities to exploit mixed precision arithmetic. Some general techniques, like multiword arithmetic (Chapter 5), memory accessors (Chapter 12), or adaptive precision kernels (Chapter 11), all can (and often have) be applied to GMRES. In this section we review strategies that are more specific to GMRES.

8.3.1 Mixed precision preconditioned GMRES

We first discuss unrestarted, preconditioned GMRES. As explained in the previous Section 8.2, the use of preconditioning can affect the stability of GMRES by introducing some dependencies on various condition numbers (of the matrix, the preconditioner, the Krylov basis, etc.). Since these condition numbers can have widely different magnitudes, this creates an opportunity to use mixed precision. One challenge is that many different (and sometimes conflicting) strategies have been proposed, depending on the precisions used for each operation and on the preconditioning style (left, right, or flexible), so it can be difficult to decide which strategy to prefer in which context. In **[40]**, we provide a comprehensive error analysis that covers virtually all possible variants and can be used to identify the numerically meaningful ones. We consider three independent precisions u_a, u_m, and u_g, corresponding respectively to the application of the matrix–vector product with A, the application of the preconditioner M, and the rest of the GMRES operations (which notably includes the orthonormalization of the basis). We use our modular framework from **[12]** and Theorem 8.4 to derive bounds on the attainable forward error for left, right, and flexible preconditioning in Theorems 3.2–3.4 of **[40]**. After operating a few simplications discussed in section 3.5 of **[40]**, these bounds become

$$\text{(left)} \qquad u_g\kappa(M^{-1}A) + u_m \max\left(\rho_M^L, \kappa(M^{-1}A)\right) + u_a\kappa(A), \tag{8.23}$$

$$\text{(right)} \qquad u_g\kappa(AM^{-1})\kappa(M) + u_m\kappa(M) + u_a\kappa(A), \tag{8.24}$$

$$\text{(flexible)} \qquad u_g\kappa(AM^{-1})\kappa(M) + u_a\kappa(A). \tag{8.25}$$

For the left-preconditioning bound, the quantity ρ_M^L can only be bounded by $\kappa(M^{-1}A)\kappa(M)$ but can be much smaller in practice due to numerical cancellation in the preconditioned matrix–vector product.

These bounds quantify the effect of the precisions u_g, u_m, and u_a on the attainable forward error. Importantly, they reveal that the multiplicative terms in front of each precision are different: for example, in bound (8.23), the term $\kappa(M^{-1}A)$ in front of u_g can be much smaller than the term $\kappa(A)$ in front of u_a. In such a configuration, we can set $u_g \gg u_a$ to balance the two terms $u_g\kappa(M^{-1}A)$ and $u_a\kappa(A)$ without altering the attainable forward error. This creates opportunities for mixed precision strategies.

As importantly, the bounds reveal significant differences between preconditioning styles (left, right, and flexible). Taking for example the terms in front of precision u_g, we have $\kappa(M^{-1}A)$ for left preconditioning instead of $\kappa(AM^{-1})\kappa(M)$ for right and flexible preconditioning. This therefore suggests that left-preconditioned GMRES might be more resilient to reductions of the precision u_g if $\kappa(M)$ is large. Conversely, the bounds suggest that right and especially flexible preconditioning are more resilient to lowering the precision u_m used for applying the preconditioner. Indeed, it is striking to observe that the flexible bound (8.25) is actually independent of u_m: this is because rounding errors incurred in the application of the preconditioner can be seen as variations in the preconditioner itself, which flexible GMRES is designed to handle. However, note that the choice of u_m will still affect the number of iterations!

Table 8.1 lists all the possible combinations of u_a, u_g, and u_m when considering the use of two distinct precisions u_{low} and u_{high}. As the table shows, the existing literature on mixed

Table 8.1: List of the mixed precision strategies presented in **[40]** with their associated dominant term in the bounds (8.23), (8.24), and (8.25) for left-, right-, and flexible-preconditioned GMRES. White cells correspond to provably meaningful strategies.

	Left	Right	Flexible
$u_a = u_g = u_m = u$	$u \max(\kappa(A), \rho_M^L)$ [231]	$u\kappa(\widetilde{A})\kappa(M)$ [231]	$u\kappa(\widetilde{A})\kappa(M)$ [230]
$u_{\text{high}} = u_a = u_m \ll u_g = u_{\text{low}}$	$u_{\text{low}}\kappa(\widetilde{A}) + u_{\text{high}} \max(\kappa(A), \rho_M^L)$ [87, 88, 90, 209, 89] **[11, 15]**	$u_{\text{low}}\kappa(\widetilde{A})\kappa(M)$ new	$u_{\text{low}}\kappa(\widetilde{A})\kappa(M)$ new
$u_{\text{high}} = u_a = u_g \ll u_m = u_{\text{low}}$	$u_{\text{low}} \max(\kappa(\widetilde{A}), \rho_M^L) + u_{\text{high}}\kappa(A)$ new	$u_{\text{low}}\kappa(M) + u_{\text{high}}\kappa(\widetilde{A})\kappa(M)$ new	$u_{\text{high}}\kappa(\widetilde{A})\kappa(M)$ [68, 166, 86, 85], **[12]**
$u_{\text{high}} = u_a \ll u_g = u_m = u_{\text{low}}$	$u_{\text{low}} \max(\kappa(\widetilde{A}), \rho_M^L) + u_{\text{high}}\kappa(A)$ new	$u_{\text{low}}\kappa(\widetilde{A})\kappa(M)$ new	$u_{\text{low}}\kappa(\widetilde{A})\kappa(M)$ new
$u_{\text{high}} = u_g \ll u_a = u_m = u_{\text{low}}$	$u_{\text{low}} \max(\kappa(A), \rho_M^L)$ new	$u_{\text{low}}\kappa(A) + u_{\text{high}}\kappa(\widetilde{A})\kappa(M)$ new	$u_{\text{low}}\kappa(A) + u_{\text{high}}\kappa(\widetilde{A})\kappa(M)$ new
$u_{\text{high}} = u_m \ll u_a = u_g = u_{\text{low}}$	$u_{\text{low}}\kappa(A) + u_{\text{high}}\rho_M^L$ new	$u_{\text{low}}\kappa(\widetilde{A})\kappa(M)$ new	$u_{\text{low}}\kappa(\widetilde{A})\kappa(M)$ new
$u_{\text{high}} = u_g = u_m \ll u_a = u_{\text{low}}$	$u_{\text{low}}\kappa(A) + u_{\text{high}}\rho_M^L$ new	$u_{\text{low}}\kappa(A) + u_{\text{high}}\kappa(\widetilde{A})\kappa(M)$ new	$u_{\text{low}}\kappa(A) + u_{\text{high}}\kappa(\widetilde{A})\kappa(M)$ new

precision GMRES has mainly focused on two strategies, that is, on two cells of this table. The first is left-preconditioned GMRES with higher precision for the products with the preconditioner and the matrix, which has been especially been used in the context of GMRES-based iterative refinement (see Section 10.2). The second is flexible GMRES with lower precision for applying the preconditioner.

Our comprehensive analysis allows for covering many new strategies, several of which are actually meaningful, in the sense that they can achieve new tradeoffs between attainable accuracy and usage of low precision. Indeed, all white cells are provably meaningful: reducing any of the three precisions would directly worsen the bound. Two new strategies that are particularly worth highlighting are:

- $u_a = u_g \ll u_m$ with right-preconditioning: this is the same combination mentioned above but with right instead of flexible preconditioning, which may be of interest to avoid doubling the Krylov basis. Our analysis reveals that right-preconditioning can be much more accurate than left-preconditioning when $\kappa(M) \ll \kappa(A)$.

- $u_a \ll u_g = u_m$ with left-preconditioning: compared with the existing combination $u_a = u_m \ll u_g$ mentioned above, this one only keeps the product with A in high precision, and switches that with M to low precision. Our analysis reveals that accuracy may be preserved when $\kappa(\widetilde{A}) \ll \kappa(A)$ and if ρ_M^L is small.

It is interesting to note that, because of the inequality $\kappa(A) \leq \kappa(\widetilde{A})\kappa(M)$, these two new strategies are meaningful in different situations: the first should be used with "lightweight" preconditioners

(for which $\kappa(\widetilde{A})$ may be large but $\kappa(M)$ is small); the second should be used with "heavyweight" preconditioners (for which the converse is true).

Next, we comment on the gray cells in Table 8.1. They correspond to strategies that cannot be proven meaningful with the theoretical error bounds alone. However, in **[40]**, we show several of them to be empirically meaningful, with varying degrees of success.

Finally, we note that in Table 8.1, we have simplified the discussion by focusing on variants that only use two distinct precisions u_{low} and u_{high}. In **[40]**, we also consider three-precision variants where u_a, u_g, and u_m are all different, and find even more meaningful strategies with new levels of tradeoff.

8.3.2 Mixed precision restarted GMRES

A popular approach to use mixed precision in GMRES is to combine it with restarting. The inner cycles of unrestarted GMRES (the for loop in Algorithm 8.2) are performed in low precision, whereas the outer loop (the while loop in Algorithm 8.2) that updates the solution and the residual is kept in high precision.

We will explain in Chapter 10 that this is actually a form of iterative refinement. We will thus discuss the behavior of this class of strategies in Chapter 10; see in particular Section 10.2.2.

8.3.3 Mixed precision relaxed GMRES

The previously presented strategies use different precisions for different operations, but these precisions are fixed throughout the iterations. Another class of approaches has focused on changing the precision across different iterations. In particular, several papers in the early 2000s showed that the precision of the matrix–vector product can be gradually lowered as the method converges. This started as a heuristic strategy based on experimental observations from Bouras and Frayssé [79], but was soon justified theoretically with the development of inexact Krylov theory—see, in particular, the works of Simoncini and Szyld [234], van den Eshof and Sleijpen [127], and Giraud, Gratton, and Langou [139].

These approaches consider an inexact matrix–vector product satisfying $w_k = (A + E_k)v_k$ (we consider the unpreconditioned case throughout the following discussion) for some error matrix E_k which depends on the iteration number k. The Arnoldi relation (8.2) becomes

$$[(A + E_1)v_1, \ldots, (A + E_k)v_k] = V_{k+1}\bar{H}_k, \tag{8.26}$$

which can be rewritten as

$$(A + G_k V_k^T)V_k = V_{k+1}\bar{H}_k, \quad \text{with } G_k = [E_1 v_1, \ldots, E_k v_k]. \tag{8.27}$$

Hence, the inexact Arnoldi relation (8.26) with A implies an exact Arnoldi relation (8.27) with a perturbed matrix $A + G_k V_k^T$. We define the true residual r_k and its inexact counterpart $\widetilde{r}_k$ as

$$r_k = b - Ax_k, \tag{8.28}$$
$$\widetilde{r}_k = b - (A + G_k V_k)x_k. \tag{8.29}$$

Since $\|\widetilde{r}_k\|$ converges to zero, the triangle inequality $\|r_k\| \leq \|\widetilde{r}_k\| + \|\widetilde{r}_k - r_k\|$ shows that $\|r_k\|$ can be guaranteed to converge by keeping the residual gap $\|\widetilde{r}_k - r_k\|$ under control. The next result, which is a rework of [139, Theorem 2], provides a concrete criterion to control this residual gap with respect to the matrix–vector perturbations E_k.

Theorem 8.6. *Let $Ax = b$ be solved with GMRES (Algorithm 8.1) and assume that the matrix–vector products with A satisfy*

$$(A + E_k)v_k = w_k, \quad \frac{\|E_k\|}{\|A\|} \leq \frac{1}{2n\kappa(A)} \min\left(1, \frac{\|b\|}{\|\widetilde{r}_{k-1}\|}\frac{\varepsilon}{2}\right), \tag{8.30}$$

with $\varepsilon > 0$ and where r_k and $\widetilde{r}_k$ are defined in (8.26)–(8.27). Then there exists $\ell \leq n$ at which $\|\widetilde{r}_\ell\| \leq \frac{\varepsilon}{2}\|b\|$, $\|\widetilde{r}_\ell - r_\ell\| \leq \frac{\varepsilon}{2}\|b\|$, and so $\|r_\ell\| \leq \varepsilon\|b\|$.

Proof. This is a slight rework of [139, Theorem 2], taking $c = \frac{1}{2}$ and $\varepsilon_c = \varepsilon_g = \frac{\varepsilon}{2}$, and dividing $\|E_k\|$ by $\|A\|$ to obtain a bound on the relative error. □

Theorem 8.6 shows that $\|E_k\|$ should be chosen to be inversely proportional to $\|\widetilde{r}_k\|$; since this quantity converges to zero as the iterations progress, this indicates that the matrix–vector product can be performed in gradually lower precision.

Note that Theorem 8.6 bounds the relative residual $\frac{\|r_\ell\|}{\|b\|}$. A similar result bounding the normwise backward error $\frac{\|r_\ell\|}{\|A\|\|x_\ell\|+\|b\|}$ is given in [139, Theorem 3].

Finally, while most of the works on relaxed GMRES have focused on reducing the precision of the matrix–vector product, we note that Gratton et al. [143, 144] have also considered the extension of this approach to reduce the precision of the inner products (that are involved in the orthonormalization of the Krylov basis).

8.4 BiCGStab

Another popular iterative method to solve general nonsymmetric linear systems is the biconjugate gradient stabilized (BiCGStab) method [247], a variant of the biconjugate gradient (BiCG) method that incorporates a stabilization step to address the erratic convergence patterns often seen in BiCG.

BiCGStab shares similarities with GMRES, since both are Krylov subspace solvers for nonsymmetric linear systems. Due to its residual minimization property, GMRES converges faster than BiCGStab. However, BiCGStab offers advantages in terms of memory usage and computational complexity compared with GMRES, as it avoids the need to store and orthonormalize a growing Krylov basis.

The right-preconditioned version of BiCGStab is described in Algorithm 8.3. At each iteration, the main computational bottleneck consists of two matrix–vector products, two applications of the preconditioner, and several vector operations (dot products and axpy). We stop the solver whenever the norm of the residual $\|r_i\|$ falls below a prescribed threshold ε, or after the maximum number of iterations $i_{\max}$ has been performed.

Similarly to GMRES, there are several strategies to exploit mixed precision arithmetic to accelerate BiCGStab. A first natural strategy is to perform the preconditioning operations in low precision. To maximize the performance, it is desirable to use low precision not only to construct the preconditioner, but also to apply it at each iteration. This, however, requires an iterative solver that can handle non-constant preconditioners, since the rounding errors incurred in the application of the preconditioner introduce variations from one iteration to the other. In the case of Krylov subspace solvers, this means that we must use a flexible formulation of the solver, as we already discussed in the case of FGMRES [230] in Section 8.3.1. In **[47]**, we make the important observation that the standard formulation of right-preconditioned BiCGStab (as outlined in Algorithm 8.3) is already flexible. This formulation was proposed by Vogel in 2007 [246]; the motivation there was to use another iterative solver as preconditioner, but it can

Algorithm 8.3 Right-preconditioned BiCGStab.

Input: $A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$, initial guess x_0, tolerance ε, maximum iteration number $i_{\max}$, right preconditioner M^{-1}.
Output: approximate solution $x_i \in \mathbb{R}^n$ to $Ax = b$.

$r_0 = b - Ax_0$
Initialize $\bar{r_0}$ arbitrarily such that $\bar{r_0}^T r_0 \neq 0$.
$\rho_0 = \bar{r_0}^T r_0$
$p_0 = r_0$
for $i = 1 : i_{\max}$ **do**
  $\widetilde{p} = M^{-1} p_{i-1}$
  $q = A\widetilde{p}$
  $\alpha = \rho_{i-1} / (\bar{r_0}^T q)$
  $s = r_{i-1} - \alpha q$
  $\widetilde{s} = M^{-1} s$
  $t = A\widetilde{s}$
  $\omega = (t^T s)/(t^T t)$
  $x_i = x_{i-1} + \alpha\widetilde{p} + \omega\widetilde{s}$
  $r_i = s - \omega t$
  if $\|r_i\| \leq \varepsilon$ **then**
    exit.
  end if
  $\rho_i = \bar{r_0}^T r_i$
  $\beta = (\alpha\rho_i)/(\omega\rho_{i-1})$
  $p_i = r_i + \beta(p_{i-1} - \omega q)$
end for

also be used to exploit low precision preconditioners. Our experiments in [47] confirm indeed that convergence is maintained even when the preconditioner is built and applied in low precision, providing a first source of acceleration. Crucially, this mixed precision variant does not require to change the rest of the algorithm and does not incur any extra cost. This is notably in contrast to GMRES, where the flexible variant FGMRES requires to double the size of the Krylov basis.

A second idea to employ mixed precision in BiCGStab is to run the entire solver in low precision, thereby also accelerating matrix–vector and dot products, and recover the accuracy by using an outer loop in high precision. Just like mixed precision restarted GMRES, this mixed precision restarted BiCGStab is once again a form of iterative refinement: we therefore postpone its discussion to Chapter 10, specifically Section 10.3.

Chapter 9

Block low-rank matrices

In many applications requiring the solution of a linear system $Ax = b$, the matrix A has a block low-rank (BLR) structure: most of its off-diagonal blocks can be well approximated by low-rank products. This property can be exploited to reduce the cost of computations with A, in particular its LU factorization. In this chapter we give an overview of some of the important questions regarding the use of BLR matrix compression: its algorithmic foundations and its asymptotic complexity, its effect on the stability of the computation, how to optimize its performance on parallel machines, opportunities to use mixed precision arithmetic, and possible extensions to achieve further complexity reductions.

The contributions described in this chapter are based on the following papers: **[8]** for Section 9.2; **[13]**, **[14]** for Section 9.3; **[16]** for Section 9.4; **[23]**, **[24]**, **[25]**, **[26]** for Section 9.5; **[28]** for Section 9.6.

9.1 Basics

A block low-rank (BLR) representation $\widetilde{A}$ of a dense matrix $A \in \mathbb{R}^{n\times n}$ has the block $q \times q$ form

$$\widetilde{A} = \begin{bmatrix} A_{11} & \widetilde{A}_{12} & \cdots & \widetilde{A}_{1q} \\ \widetilde{A}_{21} & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \vdots \\ \widetilde{A}_{q1} & \cdots & \cdots & A_{qq} \end{bmatrix}, \tag{9.1}$$

where off-diagonal blocks A_{ij} of size $n_i \times n_j$ are approximated by low-rank matrices $\widetilde{A}_{ij} = X_{ij}Y_{ij}^T$ of rank k_{ij}, where $X_{ij} \in \mathbb{R}^{n_i\times k_{ij}}$ and $Y_{ij} \in \mathbb{R}^{n_j\times k_{ij}}$.

If the ranks k_{ij} are chosen such that $\|\widetilde{A}_{ij} - A_{ij}\| \le \varepsilon\|A\|$, we obtain a BLR approximation satisfying a relative accuracy of order ε, $\|\widetilde{A} - A\| \le c\varepsilon\|A\|$, where c is a constant that depends on the choice of norm (for example, for the Frobenius norm, $c = q$). If the rank k_{ij} is small enough so that $k_{ij}(n_i+n_j) < n_in_j$, storing the low rank factors X_{ij} and Y_{ij} requires fewer entries than storing the full block A_{ij}, and A_{ij} is referred to as a low-rank block. Otherwise, it is more efficient to keep A_{ij} as is and it is referred to as a full-rank block.

Hereinafter, we denote by r the largest of the ranks k_{ij}, and we assume for simplicity that all blocks are of the same dimensions $t \times t$, so that $n = qt$. Assume that there are at most p full-rank blocks on any block-row or block-column (note that $p \ge 1$ since the diagonal blocks are

always full-rank). Then the entire BLR matrix requires storing

$$pqt^2 + 2q^2tr = ptn + 2n^2r/t \tag{9.2}$$

entries. By taking the block size $t = \sqrt{2nr/p}$, (9.2) attains its minimal value $\sqrt{2prn^3}$; this shows that the block size t should increase with both n and r. Importantly, in several applications such as the solution of certain classes of discretized partial differential equations, both p and r can be shown to be small constants independent of n [75, 74], **[23]**. Therefore, the BLR storage is proportional to $n^{3/2}$ instead of n^2 for the full uncompressed matrix. BLR compression thus leads to a reduction of the asymptotic complexity, which means that the larger the matrix, the larger the storage gain.

Algorithm 9.1 BLR LU factorization: UCF variant

{**Input**: a $q \times q$ block matrix A. **Output**: its BLR LU factors $\widetilde{L}$ and $\widetilde{U}$.}
for $k = 1$ **to** q **do**
  UPDATE:
    $A_{kk} \leftarrow A_{kk} - \sum_{j=1}^{k-1} \widetilde{L}_{kj}\widetilde{U}_{jk}$.
    for $i = k+1$ **to** p **do**
      $A_{ik} \leftarrow A_{ik} - \sum_{j=1}^{k-1} \widetilde{L}_{ij}\widetilde{U}_{jk}$ and $A_{ki} \leftarrow A_{ki} - \sum_{j=1}^{k-1} \widetilde{L}_{kj}\widetilde{U}_{ji}$.
    end for
  COMPRESS:
    for $i = k+1$ **to** q **do**
      Compute LR approximations $\widetilde{A}_{ik} \approx A_{ik}$ and $\widetilde{A}_{ki} \approx A_{ki}$.
    end for
  FACTOR:
    Compute the LU factorization $\widetilde{L}_{kk}\widetilde{U}_{kk} = A_{kk}$.
    for $i = k+1$ **to** q **do**
      Solve $\widetilde{L}_{ik}\widetilde{U}_{kk} = \widetilde{A}_{ik}$ for $\widetilde{L}_{ik}$ and $\widetilde{L}_{kk}\widetilde{U}_{ki} = \widetilde{A}_{ki}$ for $\widetilde{U}_{ki}$.
    end for
end for

The BLR representation is not only useful for reducing storage: it can also be exploited to reduce the cost of computations, and in particular that of LU factorization. Algorithm 9.1 presents the UCF variant of the BLR LU factorization, whose name stands for the order in which the three main operations are performed: update–compress–factor. Indeed, at each step k, the kth panel (composed of the kth block-row and the kth block-column) is updated with respect to the previously factored panels, which are already compressed; then, the off-diagonal blocks of the panel are compressed; and finally, the panel is factored.

Other variants are possible and have been investigated in the literature. For example, the UFC variant performs the compress after the factor, which increases the number of flops but allows for an easier handling of numerical pivoting. Conversely, the CUF variant compresses the panels at the very beginning (and thus the full uncompressed matrix never needs to be formed or stored); it however performs the updates on blocks that are already low-rank, which requires recompression operations (to prevent the ranks from growing) that are usually quite inefficient. See [49] for a detailed discussion and comparison of these variants.

A flop count analysis similar to the storage analysis in (9.2) shows that, with the same suitable choice of block size, the $O(n^3)$ flop complexity for the full LU factorization is reduced to $O(n^2r)$ for Algorithm 9.1 [23]. Moreover, these reduced storage and flop complexities for dense systems can be combined with the reductions obtained by exploiting sparsity for sparse systems. Indeed,

we can apply BLR compression to the frontal matrices in the multifrontal method (that is, by representing each of the nodes in Figure 7.3 as a BLR matrix). Then, using formula (7.1) with $\mathcal{C}_{\text{dense}}(\cdot)$ as the BLR complexity, we obtain reduced complexities reported in the second row of Table 9.1 (found in Section 9.5), under the assumption that the ranks are independent of the matrix size ($r = O(1)$). In particular, for sparse 3D problems, the BLR multifrontal method has a nearly linear $O(n \log n)$ storage complexity, and a $O(n^{4/3})$ flop complexity.

9.2 Stability

An important question is how the use of BLR compression affects the stability of LU factorization. Indeed, while the threshold ε straightforwardly controls the blockwise truncation errors, the question of how these errors propagate within the LU factorization is far from obvious. The difficulty in analyzing the stability of BLR LU factorization is twofold. First, the low-rank approximations are computed on the fly during the factorization, interlaced with the rest of the standard operations of LU factorization, and thus introduce errors during the computation. Second, the rest of these standard operations are themselves modified in order to exploit the low-rank structure of the blocks, to reduce the number of floating-point operations: for example, the product of two blocks A_{ij} and A_{jk} is computed as $X_{ij}(Y_{ij}^T X_{jk})Y_{jk}^T$ with an appropriate parenthetization to minimize the cost.

To answer this question, we develop a dedicated rounding error analysis in **[8]**, which derives bounds on the errors introduced by each of the local modified low-rank kernels, and then combines these bounds to conclude on the overall stability of BLR LU factorization. We obtain the following theorem, which proves that backward stability is maintained with respect to ε.

Theorem 9.1 (Theorem 4.3 of **[8]**). *Let $A \in \mathbb{R}^{n\times n}$ be a nonsingular matrix partitioned into q^2 blocks of order t. If Algorithm 9.1 in a precision with unit roundoff u runs to completion, it produces BLR LU factors $\widetilde{L}$ and $\widetilde{U}$ satisfying*

$$A = \widetilde{L}\widetilde{U} + \Delta A, \quad \|\Delta A\| \leq \big(\xi_q \varepsilon + \gamma_q\big)\|A\| + \gamma_c \|\widetilde{L}\|\|\widetilde{U}\| + O(u\varepsilon), \tag{9.3}$$

*where $c = t + 2r^{3/2} + q$, and where $\xi_q \leq q^2/\sqrt{6}$ (see Table 4.1 of **[8]** for a precise expression).*

This bound is to be compared with the standard LU factorization bound of order $u\|\widetilde{L}\|\|\widetilde{U}\|$; the BLR LU bound (9.3) exhibits an additional term of order $\varepsilon\|A\|$. This proves that BLR LU is backward stable and that the ε parameter can be used to reliably control the accuracy of the solution.

In **[8]** we also analyze the impact on the stability of several factors, such as the use of intermediate recompressions in the low-rank updates, the use of variants other than UCF (such as UFC), and the use of a global threshold ($\|\widetilde{A}_{ij} - A_{ij}\| \leq \varepsilon\|A\|$) as opposed to a local one ($\|\widetilde{A}_{ij} - A_{ij}\| \leq \varepsilon\|A_{ij}\|$). We find that the use of intermediate compressions, the UCF variant, and a global threshold all achieve a better compression–accuracy tradeoff, that is, they allow for significantly higher compression while only increasing the error by a modest constant factor (see values of ξ_q in Table 4.1 in **[8]**).

Theorem 9.1 on BLR LU factorization can be used to prove the backward stability of the solution of a linear system $Ax = b$ by BLR LU factorization and substitution.

Theorem 9.2 (Theorem 4.5 of **[8]**). *Let $A \in \mathbb{R}^{n\times n}$ be a nonsingular matrix partitioned into q^2 blocks of order t, and suppose that Algorithm 9.1 produces BLR LU factors $\widetilde{L}$ and $\widetilde{U}$. Let the*

system $\widetilde{L}\widetilde{U}x = b$ be solved by substitution. Then the computed solution $\widehat{x}$ satisfies

$$(A + \Delta A)\widehat{x} = b + \Delta b, \tag{9.4}$$

$$\|\Delta A\| \leq (\xi_q \varepsilon + \gamma_q)\|A\| + \gamma_{3c}\|\widetilde{L}\|\|\widetilde{U}\| + O(u\varepsilon), \tag{9.5}$$

$$\|\Delta b\| \leq \gamma_q(\|b\| + \|\widetilde{L}\|\|\widetilde{U}\|\|\widehat{x}\|) + O(u^2), \tag{9.6}$$

where $c = t + 2r^{3/2} + q$, and where $\xi_q \leq q^2/\sqrt{6}$ (see Table 4.1 of [8] for a precise expression).

9.3 Communication-avoiding BLR factorization and solve

While the reduced complexity of BLR LU factorization should in principle lead to a significantly faster execution time, translating these theoretical gains into actual speedups is quite challenging, especially in a parallel computing environment. One of the main reasons is that BLR computations tend to be memory (or communication) bound. Indeed, while BLR approximations reduce both flops and memory costs, the former are reduced by a much larger factor than the latter. This is for example reflected by the theoretical complexities for 3D regular problems (see Table 9.1): the flops are reduced by a factor of order $n^2/n^{4/3} = n^{2/3}$ whereas the memory is only reduced by a factor of order $n^{4/3}/(n \log n) = n^{1/3}/\log n$. Moreover, BLR approximations not only require partitioning the matrix into smaller blocks, but also replace these blocks by low-rank products of even smaller dimensions. The granularity of BLR computations is therefore radically smaller than standard dense linear algebra.

In order to achieve efficiency, it is therefore necessary to rethink standard algorithms in order to reduce the amount of data movement or communications. We have developed such *communication-avoiding* variants for both the BLR LU factorization and LU triangular solution.

In **[13]**, we propose two modifications to make the BLR LU factorization communication-avoiding. The first is to use a left-looking scheme rather than a right-looking one. We have explained in Section 7.1 that right- and left-looking schemes usually perform comparably in standard dense factorizations. In the BLR factorization, and more specifically the UCF variant presented in Algorithm 9.1, a key difference appears, due to the fact that the LU factors are gradually compressed, panel by panel, as the factorization advances. As illustrated in Figure 9.1, in a right-looking scheme, the entire trailing submatrix, which is uncompressed, needs to be accessed at each step to be updated. In contrast, in a left-looking scheme, we need to access instead the previously factored panels, which are already compressed. Hence, the volume of data movements is significantly lower with a left-looking scheme.

The second modification that we propose in **[13]** to improve the efficiency of the BLR LU factorization is to group together low-rank updates in order to increase their granularity. Indeed, let $\widetilde{L}_{ij} = X_{ij}Y_{ij}^T$ and $\widetilde{U}_{jk} = X_{jk}Y_{jk}^T$; the update $A_{ik} \leftarrow A_{ik} - \sum_{j=1}^{k-1} \widetilde{L}_{ij}\widetilde{U}_{jk}$ can be rewritten as

$$A_{ik} \leftarrow A_{ik} - [X_{i1} \cdots X_{i,k-1}] \begin{bmatrix} Y_{i1}^T X_{1j} & & \\ & \ddots & \\ & & Y_{i,k-1}^T X_{k-1,j} \end{bmatrix} [Y_{1j} \cdots X_{k-1,j}]^T$$

This so-called low-rank update accumulation (LUA) strategy increases the compute intensity of the BLR LU factorization. Our performance experiments with the MUMPS solver presented in **[13]** show that combining a left-looking scheme with this LUA strategy can lead to significant speedups.

In a later work, **[14]**, we extend these ideas to accelerate the BLR triangular solution step with many right-hand sides (RHS). While with a single RHS, the communication cost of the solution

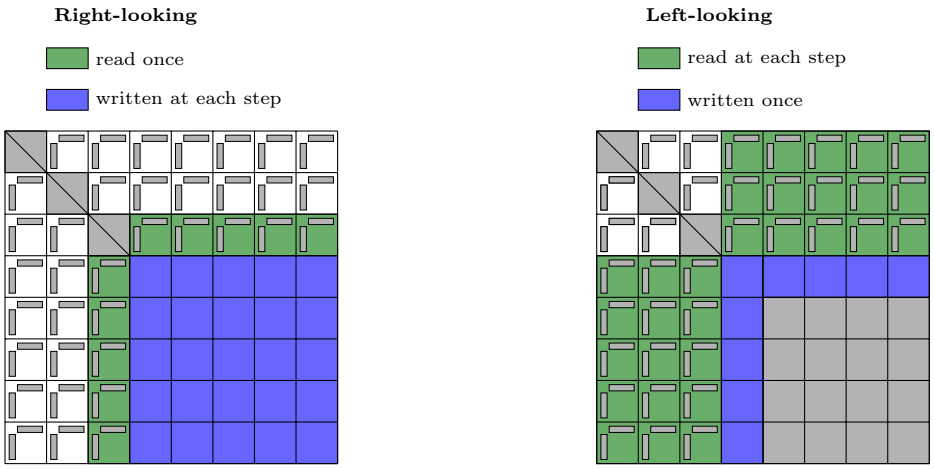


Figure 9.1: Comparison of the memory accesses in the right- and left-looking BLR LU factorizations.

step is dominated by the accesses to the compressed BLR LU factors, with many RHS, the bottleneck becomes instead the accesses to the RHS, which are uncompressed. This observation holds for both right- and left-looking schemes. Indeed, consider step k of the forward solve and denote $B \in \mathbb{R}^{n \times n_{\text{rhs}}}$ the RHS matrix, partitioned in q blocks $B_i \in \mathbb{R}^{t \times n_{\text{rhs}}}$. The right-looking scheme updates $B_i \leftarrow B_i - \widetilde{L}_{ik} B_k$ for all $i > k$, which requires accessing all the bottom part of the RHS. The left-looking scheme updates $B_k \leftarrow B_k - \widetilde{L}_{ki} B_i$ for all $i < k$, which requires accessing all the top part of the RHS. Both schemes thus share the common weakness of requiring multiple accesses to the entire RHS.

To reduce these communication costs, we propose in [14] a hybrid scheme that divides the updates $B_k \leftarrow B_k - \widetilde{L}_{ki} B_i = B_k - X_{ki} Y_{ki}^T B_i$ in two separate subtasks. We compute $W_{ki} = Y_{ki}^T B_i$ at step i, store it until step k, at which we then compute $B_k \leftarrow B_k - X_{ki} W_{ki}$. In other words, we compute the first half of the product in a right-looking scheme, but delay its second half in order to compute it in a left-looking scheme. This hybrid scheme only needs to access one block of the RHS per step, at the cost of having to store (and access) the temporary W_{ki} matrices, but these are of smaller dimensions $r \times n_{\text{rhs}}$ compared with the $t \times n_{\text{rhs}}$ RHS blocks. We carry out a communication volume analysis that shows that the hybrid scheme significantly reduces the volume of communications compared with either the right- or left-looking ones under the condition

$$\min(t, n_{\text{rhs}}) \gg 2r.$$

Thus, this hybrid scheme leads to significant gains if we have many RHS and if the LU factors are sufficiently low-rank. Our performance experiments with the MUMPS solver in [14] confirm the potential of this hybrid scheme, with which we obtained time reductions of up to 20%.

9.4 Adaptive precision BLR

In Section 6.3, we have presented an adaptive precision LRA approach developed in [16]. In the same paper, we also show how this approach can be exploited for BLR matrices. Indeed, for

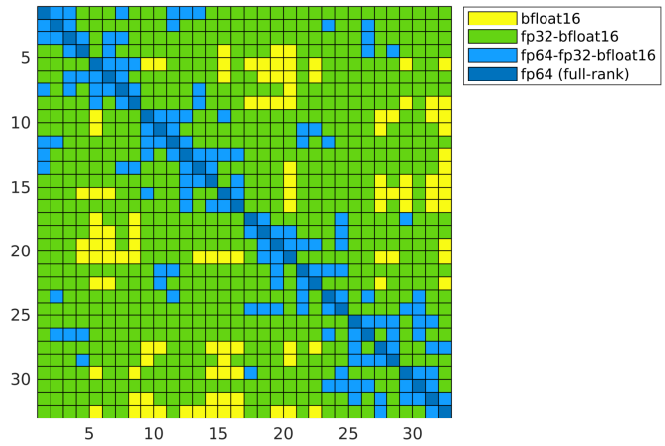


Figure 9.2: Precisions used for representing each block of a mixed precision BLR matrix with $\varepsilon = 10^{-10}$ (taken from Figure 2 in **[16]**).

each off-diagonal block A_{ij}, we can compute its adaptive precision LRA

$$\widetilde{A}_{ij} = \left[\widehat{U}_1 \dots \widehat{U}_p\right] \begin{bmatrix} \widehat{\Sigma}_1 & & \\ & \ddots & \\ & & \widehat{\Sigma}_p \end{bmatrix} \left[\widehat{V}_1 \dots \widehat{V}_p\right]^T \tag{9.7}$$

where $\widehat{U}_k$, $\widehat{\Sigma}_k$, and $\widehat{V}_k$ are stored in precision u_k and where the singular values are partitioned such that $\|\Sigma_k\| \leq \varepsilon\|A\|/u_k$. By Theorem 6.1, this yields $\|A_{ij} - \widetilde{A}_{ij}\| \leq (2p-1)\varepsilon\|A\|$. Note that this criterion and its associated error bound involve the norm of the global matrix $\|A\|$, rather than the norm of the block $\|A_{ij}\|$. As mentioned in Section 9.2, using a global criterion is indeed stable and more efficient than using a local criterion **[10]**. In the context of adaptive precision BLR, a global criterion is even more beneficial because it allows for using low precision on more entries. In fact, blocks whose norm is sufficiently small may be stored entirely in lower precision: specifically, if $\|A_{ij}\| \leq \varepsilon\|A\|/u_k$, precisions $u_1, \dots, u_{k-1}$ are not needed for this block and it may be stored directly in precisions $u_k, \dots, u_p$. This is illustrated in Figure 9.2, which shows that most blocks (those in green and yellow) do not need any entry in fp64, even for an accuracy $\varepsilon = 10^{-10}$ far smaller than the unit roundoff of fp32 arithmetic. This observation motivates a simple idea for a mixed precision block representation that uses the same precision for all entries in a given block, but different precisions for different blocks (low-rank or not). This approach is for example used by Abdulah et al. [53] (see also [54, 115]).

This adaptive precision BLR representation can do more than just reduce storage: we prove in **[16]** that the computations for the BLR LU factorization can also be performed in mixed precision. For example, consider two adaptive precision low-rank blocks $A = \sum_{k=1}^{p} X_k Y_k^T$ and $B = \sum_{k=1}^{p} U_k V_k^T$, where X_k, Y_k, U_k, and V_k are stored in precision u_k. The product AB is then given by

$$\begin{aligned} AB &= \sum_{i=1}^{p}\sum_{j=1}^{p} X_i Y_i^T U_j V_j^T \\ &= \sum_{i=1}^{p} X_i \left(\sum_{j=1}^{p} Y_i^T U_j V_j^T\right). \end{aligned}$$

Table 9.1: Flop and storage complexities of LU factorization depending on the matrix format, for $n \times n$ systems, with a constant rank bound $r = O(1)$. For the BLR^2 format we also assume a constant rank $s = O(1)$ for the shared bases. BLR$_\omega$ denotes BLR with an $O(n^\omega)$ fast matrix multiplication complexity, with $2 \leq \omega < 3$ ($\omega = \log_2 7 \approx 2.81$ for Strassen's algorithm).

	Flop complexity			Storage complexity		
	Dense	Sparse 2D	Sparse 3D	Dense	Sparse 2D	Sparse 3D
Uncompressed	$O(n^3)$	$O(n^{3/2})$	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^{4/3})$
BLR	$O(n^2)$	$O(n \log n)$	$O(n^{4/3})$	$O(n^{3/2})$	$O(n)$	$O(n \log n)$
BLR^2	$O(n^{9/5})$	$O(n)$	$O(n^{6/5})$	$O(n^{4/3})$	$O(n)$	$O(n)$
MBLR ($\ell = 2$)	$O(n^{5/3})$	$O(n)$	$O(n^{10/9})$	$O(n^{4/3})$	$O(n)$	$O(n)$
MBLR ($\ell = 3$)	$O(n^{3/2})$	$O(n)$	$O(n \log n)$	$O(n^{5/4})$	$O(n)$	$O(n)$
MBLR ($\ell = 4$)	$O(n^{7/5})$	$O(n)$	$O(n)$	$O(n^{6/5})$	$O(n)$	$O(n)$
$\mathcal{H}$	$O(n \log^2 n)$	$O(n)$	$O(n)$	$O(n \log n)$	$O(n)$	$O(n)$
$\mathcal{H}^2$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
BLR$_\omega$	$O(n^{(\omega+1)/2})$	$O(n)$	$O(n^{(\omega+1)/3})$	$O(n^{3/2})$	$O(n)$	$O(n \log n)$
BLR$_\omega$ ($\omega \approx 2.81$)	$O(n^{1.904})$	$O(n)$	$O(n^{1.27})$	$O(n^{3/2})$	$O(n)$	$O(n \log n)$

The error analysis in **[16]** shows that $S_i = \sum_{j=1}^{p} Y_i^T U_j V_j^T$ can be computed in precision $\max(u_i, u_j)$, and $\sum_{i=1}^{p} X_i S_i$ can be computed in precision u_i. We also analyze other kernels such as the product between low-rank and full-rank blocks, or the triangular solve with a low-rank block right-hand side, and show that these kernels can similarly exploit mixed precision. Overall, we prove that backward stability is maintained, with a result analogous to Theorem 9.1 with slightly different constants (Theorem 4.4 in **[16]**).

This adaptive precision BLR representation has been integrated into the MUMPS solver, where it is currently used to reduce the size of the LU factors and thus the memory cost of the solver (see Section 9.6 for an illustrative application). Reducing the size of the LU factors can also lead to time reductions for the triangular solve phase; this requires the development of an optimized memory accessor as will be described in Section 12.4.

9.5 Further reducing the complexity

As mentioned, BLR compression reduces the asymptotic storage and flop complexities of LU factorization; however, as indicated in the second row of Table 9.1, these complexities remain highly superlinear for dense matrices, and, to a lesser extent, for sparse ones arising from 3D problems. In this section, we discuss more sophisticated approaches to further reduce these complexities.

9.5.1 Hierarchical formats: taxonomy and discussion

A natural approach to reduce the asymptotic complexity is to define hierarchical representations, which use block sizes as large as possible for the off-diagonal low-rank blocks, and recursively partition the full-rank blocks (including in particular the diagonal ones) into smaller blocks. Among such hierarchical representations, the most studied is the $\mathcal{H}$ matrix format [149, 78]. We recommend the books of Hackbusch [150] and Bebendorf [74] for a comprehensive review. Several other variants have also been proposed; these different hierarchical formats can be classified based on two main criterions: whether they use weak or strong admissibility, and

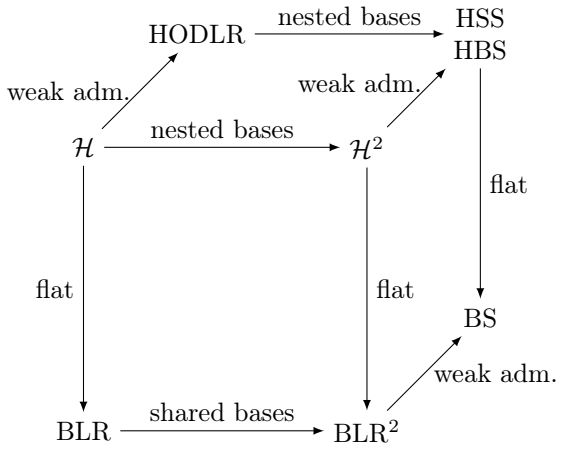


Figure 9.3: A taxonomy of structured matrix formats.

whether they use nested bases or not. Figure 9.3 summarizes this taxonomy of structured matrix formats.

Weakly admissible formats directly approximate all off-diagonal blocks as low-rank matrices, regardless of their rank, whereas strongly admissible formats recursively partition the off-diagonal blocks until their rank becomes small enough. While strongly admissible formats lead to more complex algorithms, especially for LU factorization, they can achieve better compression by partitioning blocks whose rank would be too large—in particular, the constant rank assumption $r = O(1)$ rarely holds for weakly admissible formats. $\mathcal{H}$ matrices use strong admissibility; with weak admissibility, they reduce to a format called HODLR [63].

The so-called nested bases structure consists of two main components. First, all blocks on the same block-row or block-column share the same X or Y basis, respectively. Second, these low-rank bases are nested across the levels of the hierarchy, that is, the basis at a given level is implicitly represented by the bases at the lower levels. $\mathcal{H}^2$ matrices [78] add to $\mathcal{H}$ these nested bases for an even more compact representation. The HSS [252, 94] and HBS [138] formats are weakly admissible versions of $\mathcal{H}^2$ matrices.

As indicated in Table 9.1, $\mathcal{H}$ matrices achieve quasilinear complexities already for dense problems, and the logarithmic terms can be dropped by using $\mathcal{H}^2$ matrices. As a result, both formats achieve $O(n)$ complexities for sparse problems. It is crucial to note that these formats are somewhat of an "overkill" for attaining linear complexities for sparse problems, even 3D ones. Indeed, a complexity strictly lower than $O(n^{1.5})$ for dense problems would already be enough to achieve $O(n)$ complexity for sparse 3D problems. Importantly, a lower complexity exponent is usually associated with a larger prefactor hidden in the big $O(\cdot)$, which makes $\mathcal{H}$ matrices less suitable for medium-scale dense matrices or even large scale, but sparse ones. Moreover, hierarchical formats can also be more complex to implement, especially in parallel, fully-featured, general purpose sparse solvers. For these reasons, the BLR format has been preferred in most sparse direct solvers, including MUMPS **[13]** and PaStiX [224]; STRUMPACK [136] adopts a composite approach [98] which combines the BLR format [97] with hierarchical ones, notably HSS [225], **[30]** and HODBF [192], the butterfly-based equivalent of HODLR (see Chapter 13).

Nevertheless, the superlinear complexities of BLR become a limiting factor when dealing with extreme scale matrices. This motivates the development of structured formats that achieve a middle ground between BLR and hierarchical matrices. We discuss several such ideas in the next subsections.

9.5.2 The BLR^2 format

A first idea is to exploit the nested bases structure in BLR matrices. Since BLR is a flat format with no hierarchy, the low-rank bases cannot be *nested*; however, they can still be *shared* across blocks on the same block-row or block-column. We refer to this structure as *shared bases* and to the resulting matrix format as BLR^2, since it can be understood as the $\mathcal{H}^2$ analogue to BLR matrices.

BLR^2 matrices, just like BLR ones, have a flat block $q \times q$ form defined as in (9.1). The difference is that the off-diagonal blocks $\widetilde{A}_{ij}$ are not represented independently from each other as $X_{ij}Y_{ij}^T$, but rather as $X_i C_{ij} Y_j^T$, where the bases $X_i \in \mathbb{R}^{n_i \times s_i}$ and $Y_j \in \mathbb{R}^{n_j \times s_j}$ are shared across all off-diagonal blocks on block-row i and block-column j, respectively. The $C_{ij} \in \mathbb{R}^{s_i \times s_j}$ matrices are called the coupling matrices; if the shared bases X_i and Y_j are built to have orthonormal columns, then C_{ij} contains all the spectrum information of the $\widetilde{A}_{ij}$ block and can therefore be represented as a rank-r_{ij} product $C_{ij} = \Phi_{ij}\Psi_{ij}^T$.

Gillman et al. [138] describe a flat version of their HBS format, called block separable (BS), which is a weakly admissible variant of BLR^2. However, BS was only presented as a first step towards HBS, and not studied in its own right. In **[25]**, we carry out a dedicated study of the BLR^2 format to assess the potential of exploiting shared bases in BLR matrices. We propose algorithms to construct a BLR^2 matrix, both from a dense one and from one already compressed as BLR. We also propose LU factorization and solution algorithms in order to solve linear systems.

Assume for simplicity that the ranks of the coupling matrices are all equal to $r = O(t^\alpha)$ and that the size of the shared bases are all equal to $s = O(t^\beta)$, where t is the block size and $\alpha, \beta \geq 0$. In **[25]**, we show that the BLR^2 LU factorization achieves the following asymptotic complexities:

$$\text{Storage} = O(n^{\frac{4-\alpha-\beta}{3-\alpha-\beta}}), \tag{9.8}$$

$$\text{Flops} = O(n^{\frac{9-\alpha-2\beta}{5-\alpha-2\beta}}). \tag{9.9}$$

In particular, if the ranks r and s are both $O(1)$ constants (that is, $\alpha = \beta = 0$), then we obtain complexities in $O(n^{4/3})$ for storage and $O(n^{9/5})$ for flops, which represents an asymptotic improvement over standard BLR, as indicated in Table 9.1.

However, our experiments on a range of real-life matrices suggest that the ranks of the shared bases are rarely constant, and can on the contrary become quite large if too many blocks are forced into the shared bases, such as when weak admissibility is used. We propose a hybrid strategy in which the blocks of rank larger than some $r_{\max}$ are kept separately, outside the basis (under BLR form, that is, either low-rank or dense), so that only the remaining blocks of small rank share their bases. The relevance of this approach is illustrated in Figure 9.4, which plots the storage cost for various matrices depending on $r_{\max}$. As we increase $r_{\max}$, more and more blocks share their bases; the figure shows that the optimal storage is always significantly lower than the storage for either of the two extreme strategies (no shared bases with $r_{\max} = 0$ or all blocks share their bases with $r_{\max} = \infty$).

9.5.3 The MBLR format

As explained previously, hierarchical formats recursively partition full-rank blocks in order to reduce the asymptotic complexity. In the $\mathcal{H}$ format and its variants, this recursive partitioning is applied until the full-rank blocks become smaller than a given constant independent of the matrix size; thus, there are about $\log_2 n$ levels of hierarchy in an $\mathcal{H}$ matrix of order n.

In **[24]**, we propose a format that we call multilevel BLR (MBLR), which is based on the idea of using a fixed number of levels ℓ instead. This is motivated by the fact that, as mentioned

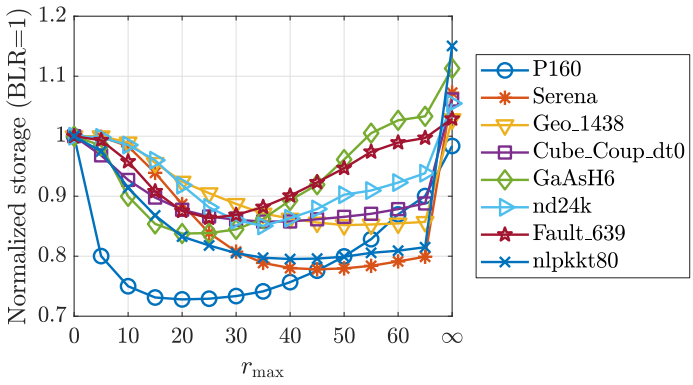


Figure 9.4: Storage for strongly admissible BLR^2 format depending on the maximal rank value $r_{\max}$ allowed in the shared bases. Storage is normalized with respect to BLR format ($r_{\max} = 0$). Weakly admissible BLR^2 corresponds to $r_{\max} = \infty$. We have used $\varepsilon = 10^{-8}$.

in Section 9.5.1, a deeper hierarchy usually leads to a larger prefactor term in the complexity, is more complex to implement, and, most importantly, is not needed to achieve $O(n)$ complexity for sparse problems. The main question that we elucidate is therefore: how many levels ℓ are actually necessary to achieve linear complexity for sparse problems?

To give an intuition of how the MBLR format works, let us consider to the two-level case ($\ell = 2$). We consider a block $q \times q$ matrix partitioned into $t \times t$ blocks, and we assume that there are at most $p = O(1)$ full-rank blocks on any block-row or block-column. Moreover, for simplicity, we also assume that the remaining blocks are of rank at most $r = O(1)$. Then, with a two-level BLR format, each of the full-rank blocks is compressed as a (one-level) BLR matrix. Hence, its storage cost is in $O(t^{3/2})$, according to the one-level BLR complexity. The storage complexity is therefore of order

$$qt^{3/2} + q^2 t = nt^{1/2} + n^2/t,$$

and for $t = n^{2/3}$, this expression attains its minimal value $O(n^{4/3})$. We conclude that using two levels reduces the asymptotic storage complexity from $O(n^{3/2})$ to $O(n^{4/3})$.

The flop complexity can also be reduced using more levels. To do so, we must adapt the BLR LU factorization described in Algorithm 9.1 to the MBLR format. The algorithm is essentially the same as presented in Algorithm 9.1, except that the kernels now involve MBLR blocks rather than full-rank ones. For example, the line that computes the LU factorization $\widetilde{L}_{kk}\widetilde{U}_{kk} = A_{kk}$ now consists in a (recursive) MBLR LU factorization, rather than a dense one. Similarly, the line that solves $\widetilde{L}_{kk}\widetilde{U}_{ki} = \widetilde{A}_{ki}$ now takes the form of an MBLR triangular solve since $\widetilde{L}_{kk}$ is an MBLR matrix. Moreover, with strong admissibility (that is, off-diagonal blocks are allowed to be MBLR), further kernels are necessary, such as an MBLR triangular solve with an MBLR block right-hand side, and block products between any combination of low-rank and MBLR blocks. We omit a detailed description of these kernels, which can be found in **[24]**.

In **[24]**, we carry out a recursive analysis to compute the asymptotic complexity for a general number of levels ℓ, both in terms of storage and flops. We obtain the following result.

Theorem 9.3 (Theorem 5.1 in **[24]**)**.** *Let us consider a dense matrix of order n represented as an ℓ-level MBLR matrix and let r be a bound on the maximal rank of any block on any level.*

Then the storage and flop complexities of the MBLR LU factorization are

$$\mathit{Storage} = O\big(n^{\frac{\ell+2}{\ell+1}}\, r^{\frac{\ell}{\ell+1}}\big), \tag{9.10}$$

$$\mathit{Flops} = O\big(n^{\frac{\ell+3}{\ell+1}}\, r^{\frac{2\ell}{\ell+1}}\big). \tag{9.11}$$

Theorem 9.3 proves that the asymptotic complexity for dense MBLR matrices steadily decreases as the number of levels increases. Table 9.1 reports the resulting complexities for $\ell = 2, 3, 4$, under the assumption that $r = O(1)$. Crucially, this confirms that only a few levels are enough to achieve linear complexities for sparse problems. Even for 3D problems, two levels suffice to achieve $O(n)$ storage complexity, and three levels suffice to achieve $O(n \log n)$ flop complexity. We confirm these complexity bounds experimentally in **[24]**.

9.5.4 Fast matrix multiplication

A completely different idea to improve the asymptotic flop complexity is to consider the use of fast matrix multiplication. Indeed, all previously given complexity bounds assume the use of conventional matrix multiplication, whose complexity for $n \times n$ matrices is in $O(n^3)$. So-called "fast" algorithms can reduce this complexity to $O(n^\omega)$, with $2 \leq \omega < 3$. The first and most well-known one is Strassen's algorithm [239], for which $\omega = \log_2 7 \approx 2.81$.

Pernet and Storjohann [223] investigate the use of fast matrix multiplication for accelerating computations on HODLR (that is, weakly admissible $\mathcal{H}$) matrices. Since the complexity is already linear in n in the hierarchical case, their goal is to reduce the asymptotic dependence on r. They show that exploiting fast matrix multiplication for each intermediate product in the $\mathcal{H}$ factorization can reduce its asymptotic complexity from $\widetilde{O}(nr^2)$ to $\widetilde{O}(nr^{\omega-1})$ (we hide logarithmic factors in the $\widetilde{O}(\cdot)$ notation for simplicity).

Can fast matrix multiplication similarly be used with BLR matrices? In this case, the complexity $O(n^2 r)$ is linear in r; it is its quadratic dependence on n that we would like to reduce. We investigate this question in **[26]**. We first show that accelerating each intermediate product separately as in [223] only reduces the $O(n^2 r)$ BLR complexity to $O(n^2 r^{\omega-2})$. This is not a satisfactory result, since only the asymptotic dependence on r is reduced, instead of that on n. We explain in **[26]** that this is because the fast multiplication of rectangular matrices reduces the complexity exponent associated with the smallest of the three dimensions. Unfortunately, the flop bottleneck of the BLR factorization consists of multiplying $b \times b$ blocks of rank r with $r \ll b$, and hence involves highly rectangular matrix products.

To overcome this obstacle, we devise a new BLR factorization algorithm (Algorithm 4.1 in **[26]**) that concatenates low-rank bases across a given block-row or block-column together. This is similar to the construction step for building a BLR^2 matrix from a BLR one that we proposed in **[25]**: the low-rank bases are concatenated before being compressed to find a shared basis. Here, we only concatenate them in order to increase the granularity of the low-rank operations. This allows for better taking advantage of fast matrix arithmetic and successfully reduces the asymptotic dependence on n of the complexity. We prove indeed in Theorem 4.1 of **[26]** that the complexity is reduced to $O(n^{(\omega+1)/2} r^{(\omega-1)/2})$. This is a significant asymptotic reduction, even for practical values of ω. For example, for Strassen's algorithm ($\omega = \log_2 7 \approx 2.81$), we obtain a complexity approximately in $O(n^{1.904} r^{0.904})$.

A possible drawback of fast matrix arithmetic is its larger error bounds in floating-point arithmetic [159], [162, sect. 23.2]. Interestingly, this problem is less visible in the BLR setting. Indeed, Theorem 9.1 shows that the low-rank truncation errors (of order ε) combine additively with the floating-point rounding errors (of order u), where usually $u \ll \varepsilon$. Since the use of fast arithmetic only concerns rounding errors, we can expect it to have a limited impact on the accuracy of the BLR LU factorization. This is for example illustrated in Figure 3.1(B) of **[8]**.

9.5.5 Triangular solves with sparse right-hand side or sparse solution

So far we have been concerned with the complexity of the LU factorization. The complexity of the LU triangular forward and backward solves is also relevant in contexts where its cost is significant, such as when dealing with multiple right-hand sides or when using BLR as a preconditioner for iterative methods.

In general, the number of flops required for the triangular solves is proportional to the number of entries in the LU factors, and hence its asymptotic flop complexity is the same as the storage complexity. However, complexity reductions can be obtained when dealing with a sparse right-hand side b (for the forward solve $Ly = b$) or a sparse solution x (for the backward solve $Ux = y$). Such sparse b and x arise in several applications, such as in geosciences for seismic imaging **[31, 32]**, [61].

In the context of sparse direct solvers such as the multifrontal method (see Section 7.2), the forward and backward solves amounts to bottom-up and top-down traversals of the multifrontal tree, respectively, where at each node, a dense triangular solve is performed. However, a node only needs to be traversed if it is on the path between the root node and a node associated with a nonzero coefficient in b or x. Therefore, when b or x are sparse, entire parts of the tree can be skipped. As an extreme example, if b or x only have one nonzero element, it is sufficient to traverse the branch connecting that nonzero element to the root.

Naturally, skipping part of the tree affects the complexity calculations given in (7.1) for regular problems reordered with nested dissection. Crucially, the asymptotic complexity reduction obtained by exploiting the sparsity of b or x strongly depends on dense storage complexity $\mathcal{C}_{\text{dense}}$ of each front; the reductions are indeed significantly larger for compressed formats such as BLR. Intuitively, this is because without BLR compression, the large fronts at the top of the tree tend to dominate the total complexity, and so removing branches from the tree is less impactful. In contrast, with BLR compression, the relative storage of the bottom of the tree increases, and so exploiting the right-hand side or solution sparsity becomes even more beneficial.

This phenomenon can be quantified more precisely by using some sparsity model for b and x. We have carried out the following complexity analysis in the context of the PhD thesis of Moreau [200, chap. 2]; it has been presented at some conferences without proceedings such as **[50]** but was never published. Let us consider a 3D problem ($d = 3$ in (7.1)) of order $n = N^3$. Let p denote the number of nonzeros of b or x. Clearly, if $p = O(1)$ (that is, if p is a constant independent of n), then the solves requires traversing $O(1)$ branches of the tree, which yields the complexity formula

$$\sum_{\ell=0}^{\log_2 N} \mathcal{C}_{\text{dense}}\Big(\frac{N^2}{2^{2\ell}}\Big). \tag{9.12}$$

If $\mathcal{C}_{\text{dense}}(m) = O(m^2)$ (no compression), we recover the same $O(n^{4/3})$ solve complexity as when b or x are dense. In this case, the sparsity of b and x may reduce the solve time, but not its *asymptotic* complexity. In contrast, let $\mathcal{C}_{\text{dense}}(m) = O(m^{3/2})$ (BLR compression with constant ranks). Now, (9.12) yields $O(n)$ complexity, to be compared to the $O(n \log n)$ BLR solve complexity for dense b or x. We can therefore expect the solve time reduction to increase as $\log_n$. Interestingly, lower dense complexities lead to even higher asymptotic improvements. For example, with the two-level MBLR format (see Section 9.5.3), we have $\mathcal{C}_{\text{dense}}(m) = O(m^{4/3})$ and hence (9.12) yields an $O(n^{8/9})$ complexity, which is *sublinear* in n, and a factor $n^{1/3}$ smaller than the two-level MBLR solve $O(n)$ complexity for dense b or x.

This analysis extends to a number of nonzeros $p = O(n^\beta)$ that depends on n; this can certainly happen, for example, when the nonzeros correspond to a 2D plane in a 3D domain, in which case $\beta = 2/3$. In such cases, it is useful to distinguish where the p nonzeros are located

geometrically, since it affects the resulting asymptotic complexity. In the worst case, they are uniformly distributed in the domain, and so the greater p is, the more branches need to be traversed; in this case, the asymptotic complexity directly depends on p and the asymptotic reduction obtained from the sparsity of b and x might be less than when $p = O(1)$. However, in typical applications with sparse b and x, such as in geosciences, the nonzeros correspond to physical locations surrounding a given so-called "source" point in the domain, and so are geometrically clustered. As a result, the number of branches to be traversed remains small and the asymptotic reduction from exploiting the sparsity of b and x might be the same as when $p = O(1)$. Specifically, for 3D domains, assuming a dense complexity $\mathcal{C}_{\text{dense}} = O(m^\alpha)$ with $\alpha < 3/2$, the asymptotic complexity of the solve with $p = O(n^\beta)$ clustered nonzeros is the same as with $O(1)$ nonzeros for any $\beta \leq 2\alpha/3$ [200].

Finally, the above complexity analysis for a single vector b or x also extends to multiple right-hand sides B and solutions X. Because the nonzero structure of each column of B and X is potentially different, we must pay attention to how we handle such cases since a naive implementation would require traversing the nodes corresponding to the union of the sparsity patterns across all columns. Naturally, one option is to process each column sequentially but this prevents the use of efficient BLAS-3 operations. A better approach is to process the columns by blocks, and permute them so that columns that belong to the same block share a similar sparsity pattern; see the study of Amestoy et al. [62] for various such strategies.

9.6 An illustrative application: full waveform inversion

We conclude this chapter by illustrating the impact on applications of the BLR compression (and its adaptive precision variant), which we have integrated in the MUMPS sparse direct solver **[13]**. The results presented here were obtained in the context of a collaboration with the WIND project dealing with full waveform inversion (FWI) applications; see **[28]**.

High resolution imaging has many applications in civil engineering, oil exploration, earthquake seismology, medical imaging, nondestructive control testing, etc. FWI has become the baseline imaging method in many application domains with the development of new acquisition technologies and the continuous advances of high-performance computing. FWI relies on simple principles: it aims at estimating the constitutive parameters of a medium contained in the coefficients of the wave equation by minimizing a distance between waves recorded by an array of sensors and their numerically simulated counterparts. As such it is a PDE-constrained optimization problem, which is solved with iterative gradient-based methods. At each iteration, building the gradient of the misfit function with the adjoint-state method requires to compute the incident wavefields triggered by each source of the experiment and the so-called adjoint wavefields by propagating the data residuals backward in time from the sensors. The wave equation is classically solved with numerical methods such as finite difference or finite element schemes to exploit the full information content in the data. Moreover, the wave equation can be solved either in the time-space or in the frequency-space domain. In the former case, solving the wave equation is an initial condition evolution problem, which can be solved with explicit (matrix-free) schemes. In the latter case, solving the time-harmonic wave equation for each source of the experiment is a boundary value problem requiring the solution of a large and sparse complex-valued linear system with multiple sparse right-hand sides. Operto and al. [213] have developed a frequency-domain FWI for geophysical applications to image a geological medium containing tens to hundreds of millions of unknowns and hundreds to thousands of seismic sources (right-hand sides). They implement FWI in the frequency domain, where the Helmhotz equation is solved with a finite difference method and the MUMPS solver.

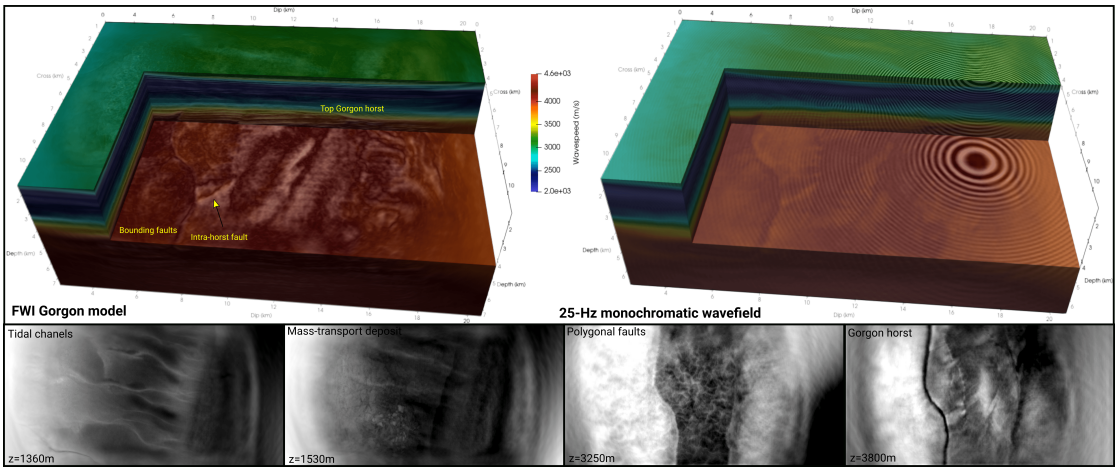


Figure 9.5: Seismic imaging of the Gorgon field by frequency-domain Full Wavform Inversion (FWI) using MUMPS solver. Top left: Reconstructed medium parametrized by compressional wavespeed. Top right: Same as previous figure with 25-Hz monochromatic wavefield superimposed. Bottom panels: depth slices of the medium reconstruted by FWI. From left to right, tidal channels, mass-transport deposits, polygonal faults, and reservoir partitioned into fault blocks. At 25 Hz, the matrix contains 531 million unknowns.

Table 9.2: Results for the Gorgon field. FR: standard full-rank MUMPS solver in fp32; BLR: block low-rank solver with $\varepsilon = 10^{-5}$; +mixed: adaptive precision variant (Section 9.4) using three precisions (fp32, rp24, rp16 from Table 12.1) for storage. Ana.+Fac.: analysis+factorization time; Solve: time to compute solution with one right-hand side. Results obtained on Adastra supercomputer using 48 000 cores (500 MPI × 96 threads/MPI).

LU size (TBytes)			Flops		Time (s)		Backward error
FR	BLR	+mixed	FR	BLR+mixed	BLR+mixed		BLR+mixed
					Ana.+Fac.	Solve	
73	34	26	2.6×10^{18}	0.5×10^{18}	5946	27	7×10^{-4}

In **[28]**, we show how the memory and computational time of the LU factorization can be reduced with BLR approximations and their adaptive precision variant, which allows for tackling large scale (high frequency) problems that would be unfeasible with the standard full-rank (FR) solver. Figure 9.5 presents the Gorgon gas field, an industrial seabed case study from the North-West continental shelf, Australia. The matrix to be solved is a complex matrix with 531 million unknowns. Table 9.2 reports the corresponding costs. The number of flops for the FR LU factorization is 2.6×10^{18}, and the estimated memory necessary for the FR LU factors in fp32 is equal to 73 TeraBytes. This is an estimate because the FR factorization could not actually be carried out due to the large memory requirement. Here fp32 arithmetic is the baseline because it provides sufficient accuracy for this class of applications.

The results in Table 9.2 show that BLR approximations with a threshold $\varepsilon = 10^{-5}$ reduce the size of the LU factors by a factor over 2×. Moreover, lower precisions can be used via the adaptive precision approach described in Section 9.4. Here, the BLR+mixed approach uses three precisions: fp32, and two 24- and 16-bit custom formats with truncated significand, rp24 and rp16, that will be defined in Table 12.1 in Chapter 12. The use of adaptive precision BLR further reduces the LU size by 25%, bringing the total reduction factor to almost 3× compared with the fp32 FR baseline. Moreover, the backward error remains satisfactory and of order ε, as guaranteed by our analysis.

BLR approximations also reduce the flops for the LU factorization by a factor over 5× with

respect to the FR baseline. Here, mixed precision is only used for storage, and does not affect the flop count. It may introduce a small time overhead due to the need to convert between custom and fp32 precisions, but this overhead is typically very small for the factorization. For the solve, which is a memory-bound operation, we may hope that the reduced LU size in mixed precision would translate into a reduced time, but this requires developing an efficient memory accessor that we will describe in Section 12.4. Note that the results in Table 9.2 predate the development of this accessor and so the indicated solve time may be further optimized. This is an important point because hundreds to thousands of right-hand sides need to be processed, so the FWI simulation can easily be dominated by the time for solution, which thus becomes more critical that the time for factorization.

These results have been achieved on the Adastra supercomputer at CINES (Montpellier, France), using 48,000 cores (500 MPI processes $\times$ 96 threads/MPI). While the number of cores used is mainly dictated by the memory requirements of the solver, it is interesting to measure how efficiently we make use of them. Each node of Adastra is equipped with two 96-core AMD Genoa EPYC 9654 processors, running at 2.4 GHz, which leads to a theoretical FLOPS rate of 38.4 GFLOPS per core in fp64 arithmetic. This yields a total theoretical peak of 3686 TFLOPS on 48,000 cores using fp32 arithmetic. The actual FLOPS rate of the BLR LU factorization is obtained by dividing the $0.5 \times 10^{18} \times 4$ complex fp32 flops by the time, yielding about 336 TFLOPS. This corresponds to 9% of the theoretical peak, which is quite satisfactory considering the large number of cores and the use of BLR approximations, which reduce the absolute time but significantly degrade the arithmetic intensity and thus the FLOPS rate as explained in Section 9.3. In fact, if we take instead the FR flops as baseline to compute an "effective" TFLOPS metric, we nearly obtain an impressive 50% of the peak.

Chapter 10

Iterative refinement

Iterative refinement (IR) for solving a linear system $Ax = b$ is perhaps the oldest mixed precision algorithm, and certainly one of the most successful ones. It was programmed by Wilkinson in 1948 and since then it has been extensively analyzed and discussed.

In its most general form, IR simply repeats the following three steps to refine a given initial approximate solution x_0 to $Ax = b$.

1. $r = b - Ax_0$
2. Solve $Ad = r$
3. $x_1 = x_0 + d$

This corresponds to Newton's method to find the zero of the function $f(x) = b - Ax$; in fact, IR converges in one iteration if step 2 is solved exactly. The difficulty, but also what makes this algorithm interesting to analyze, is that rounding errors delay its convergence.

IR is a versatile method that can take many different forms depending on both the precisions used for each of its steps, and the solver used to compute the correction term d. In this chapter, we review some of the most popular variants that use direct LU factorization–based methods, Krylov methods such as GMRES, or their combination.

The contributions described in this chapter are based on the following papers: **[5]**, **[27]**, **[15]** for Section 10.1; **[11]**, **[15]**, **[17]** for Section 10.2; **[47]** for Section 10.4. The discussion on the historical developments of IR are based on the survey paper **[10]**; see also [245, Chap. 3].

10.1 LU factorization–based iterative refinement (LU-IR)

Algorithm 10.1 presents a version of IR based on an LU factorization of A, hereinafter referred to as LU-IR. In its most general form, LU-IR uses three precision parameters. The LU factorization is computed in precision u_f and is used to compute the initial solution x_0 by substitution in precision u_f. Moreover, the LU factors are then reused at each step of the refinement loop to solve for the correction term d by substitution, also in precision u_f. The residual is computed in precision u_r, and the solution x is stored and updated in the working precision u.

Carson and Higham [88] analyze this three-precision form of LU-IR and obtain the following result, in which the componentwise condition number

$$\mathrm{cond}_\infty(A, x) = \frac{\||A^{-1}||A||x|\|_\infty}{\|x\|_\infty}$$

appears. (Note that $\mathrm{cond}_\infty(A, x) \le \kappa_\infty(A)$.)

Algorithm 10.1 LU-IR

Input: $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$.
Output: an approximate solution $x \in \mathbb{R}^n$ to $Ax = b$.

1: Compute the factorization $PA = LU$ in precision u_f.
2: Solve $LUx_0 = Pb$ by substitution in precision u_f.
3: **for** $i = 0$ **to** $i_{\max}$ or until converged **do**
4: Compute $r_i = b - Ax_i$ in precision u_r.
5: Solve $LUd_i = Pr_i$ by substitution in precision u_f.
6: Update $x_{i+1} = x_i + d_i$ in precision u.
7: **end for**

Theorem 10.1. *Let LU-IR (Algorithm 10.1) be applied to a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is nonsingular, and let $\widehat{L}$, $\widehat{U}$ be the computed LU factors. If $\phi = \||A^{-1}||\widehat{L}||\widehat{U}|\|_\infty u_f$ is sufficiently less than 1 then at each refinement step the forward error is reduced by a factor ϕ until an iterate $\widehat{x}$ is produced for which*

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \le 4pu_r \operatorname{cond}_\infty(A, x) + u, \tag{10.1}$$

where p is the maximum number of nonzeros in any row of $[A\ b]$.

Proof. This is a direct application of [88, Corollary 3.3] to LU-IR, using [88, Eq. (7.2)]. □

Theorem 10.1 captures two essential properties of LU-IR. First, it shows that the limiting accuracy (10.1) is independent of the precision u_f, which only affects the convergence of the algorithm; this motivates the use of lower precision for the LU factorization. Second, it shows that the limiting accuracy depends on the condition number $\operatorname{cond}_\infty(A, x)$ only via the precision of the residual u_r, but not the working precision u; this motivates the use of higher precision for the residual. Thus, in practice, the combinations of interest for the precision parameters satisfy $u_r \le u \le u_f$.

10.1.1 Historical developments

IR was programmed on a digital computer by Wilkinson in 1948 [249, p. 111 ff.]. The traditional form of IR used by Wilkinson and others used a stable LU factorization with partial pivoting, with $u_f = u$ and $u_r = u^2$. With such a stable LU factorization and with $u_f = u$, the convergence condition $\phi \ll 1$ in Theorem 10.1 reduces to $\kappa(A)u \ll 1$. Thus, as long as the problem is not numerically singular with respect to the working precision, this form of LU-IR will converge. The use of extended precision in the computation of the residual is motivated by the limiting accuracy (10.1). Indeed, if we set $u_r = u^2$, we obtain a limiting accuracy of order u, independent of the conditioning of the problem as long as $\operatorname{cond}_\infty(A, x)u \le 1$. Wilkinson, and subsequent authors, took advantage in computing the residual of the ability of many machines of the time to accumulate inner products at twice the working precision at little or no extra cost. The method was also used by Wilkinson and colleagues on desk calculating machines, making use of their extra length accumulators in computing residuals [130].

In the 1970s, another usage of IR came to the fore: fixed precision refinement, in which only one precision $u = u_f = u_r$ is used. This is motivated by the term $\||A^{-1}||\widehat{L}||\widehat{U}|\|_\infty$ in Theorem 10.1, which can be much larger than $\kappa_\infty(A)$ if the LU factorization is unstable. This is notably the case when a weaker form of pivoting is used to accelerate the factorization and/or

preserve the sparsity of the matrix (see Section 10.1.3). Other types of unstable, but faster, factorizations have been combined with iterative refinement to remedy their instability, such as incomplete LU factorization [263] or Cholesky factorization for quasidefinite systems [137]. Jankowski and Woźniakowski [171] proved that an arbitrary linear equation solver is made normwise backward stable by the use of fixed precision IR, as long as the solver is not too unstable to begin with and A is not too ill conditioned. Skeel [235] analysed fixed precision IR for LU factorization with partial pivoting and showed that one step of refinement yields a small componentwise backward error under suitable conditions. Higham [160] extended the componentwise backward error analysis of fixed precision IR to a general solver, and later gave an analysis [161] that covers the traditional and fixed precision forms and a general solver.

In the 2000s, hardware emerged in which fp32 arithmetic was much faster than fp64 arithmetic, such as Intel chips with SSE instructions (a factor about 2) and the Sony/Toshiba/IBM (STI) Cell processor (a factor up to 14) [179]. Motivated by this speed difference, Langou et al. [181] proposed a new usage of IR in which the LU factors are computed at a precision *lower* than the working precision (specifically, single versus double precision), that is, $u_f > u$. If the LU factorization algorithm is numerically stable, convergence is guaranteed provided that $\kappa(A)u_f \ll 1$. This approach is attractive because most of the work ($O(n^3)$ flops for dense systems) is done in the factorization phase; the refinement phase ($O(n^2)$ flops per iteration) has negligible cost for large n, as long as the number of iterations remains reasonable. Thus, asymptotically, we may expect the speed of the entire solution to be determined by the speed of the lower precision arithmetic. Using the Cell processor, Langou et al. [181] solved linear systems with double precision accuracy, but speedups of up to a factor 8 over a double precision factorization, by using LU-IR with a single precision factorization. Further experimental results are reported by Buttari et al. in [84] for dense linear systems and in [83] for sparse ones. See [70] for an overview of the methods developed in this period.

The popularity of LU-IR with a lower precision factorization has grown again in the recent years with the emergence of half precision arithmetic (fp16 and bfloat16). Indeed, half precision arithmetic can be much faster on some hardware, notably GPU accelerators. However, LU-IR with a half precision factorization can only guarantee convergence for well-conditioned problems: the condition $\kappa(A)u_f \ll 1$ translates to $\kappa(A) \ll 2000$ in fp16 and $\kappa(A) \ll 300$ in bfloat16. Two main approaches have been proposed to extend the applicability of half precision iterative refinement to a wider range of problems: the first uses a more accurate solver, such as GMRES, for computing the correction term d (see Section 10.2); the second uses more accurate hardware such as tensor cores (see the following Section 10.1.2).

10.1.2 LU-IR with GPU tensor cores

As discussed in Section 4.1, GPU tensor cores and other similar units satisfying Model 4.1 can be much more accurate than standard half precision floating-point units by using single precision for the intermediate accumulations in matrix products. In practical implementations of LU factorization (see Algorithms 7.1 and 7.2), the matrix is partitioned by blocks so that the LU factorization mostly relies on matrix products. Thus, such tensor cores units can also be used to improve the accuracy of LU factorization. Haidar et al. [153] propose a right-looking LU factorization algorithm (Algorithm 7.2) that harnesses mixed precision fp16/fp32 tensor cores to accelerate the updates of the trailing submatrix (the UPDATE step), which account for the $O(n^3)$ part of flops; the remaining $O(n^2)$ flops (the FACTOR step) are carried out in fp32 arithmetic.

In **[5]**, we analyze this algorithm and we obtain the following results.

Theorem 10.2 (Theorem. 4.3 of **[5]**). *Let the partitioned LU factorization (Algorithm 7.2) of $A \in \mathbb{R}^{n\times n}$ be computed using precision u_{high} for the* FACTOR *step and using a mixed precision matrix multiplication unit satisfying Model 4.1 for the* UPDATE *step. Then the computed LU factors $\widehat{L}$ and $\widehat{U}$ satisfy*

$$\widehat{L}\widehat{U} = A + \Delta A, \quad |\Delta A| \leq \left(2u_{\text{low}} + u_{\text{low}}^2 + \gamma_n^{\text{high}}(1+u_{\text{low}})^2\right)\left(|A| + |\widehat{L}||\widehat{U}|\right), \tag{10.2}$$

where $\gamma_n^{\text{high}} = nu_{\text{high}}/(1-nu_{\text{high}})$.

Theorem 10.3 (Theorem. 4.4 of **[5]**). *Let the partitioned LU factorization (Algorithm 7.2) of $A \in \mathbb{R}^{n\times n}$ be computed using precision u_{high} for the* FACTOR *step and using a mixed precision matrix multiplication unit satisfying Model 4.1 for the* UPDATE *step. Moreover let the system $\widehat{L}\widehat{U}x = b$ be solved by substitution in precision u_{high}. Then the computed solution $\widehat{x}$ satisfies*

$$(A+\Delta A)\widehat{x} = b, \quad |\Delta A| \leq \left(2u_{\text{low}}+u_{\text{low}}^2+\gamma_n^{\text{high}}(1+u_{\text{low}})^2+2\gamma_n^{\text{high}}+(\gamma_n^{\text{high}})^2\right)\left(|A|+|\widehat{L}||\widehat{U}|\right), \tag{10.3}$$

where $\gamma_n^{\text{high}} = nu_{\text{high}}/(1-nu_{\text{high}})$.

Theorems 10.2 and 10.3 show that the reduced error bound (of order $u_{\text{low}} + nu_{\text{high}}$ instead of nu_{low}) for matrix multiplication in Theorem 4.1 is inherited by both LU factorization and the solution of linear systems, respectively. This error bound reduction translates into a significant accuracy boost in practice: for example, Figure 4.2 in **[5]** shows that this mixed precision LU factorization algorithm achieves a reduction of the backward error by about two orders of magnitude compared with a standard fp16 factorization. Therefore, using it in LU-IR leads to a faster convergence and allows for handling a wider range of condition numbers (see for example [153, Figure 7(b)]).

One weakness of this approach is that the matrix is stored in fp32 arithmetic, and thus the reduced memory consumption and data movement of fp16 arithmetic are not exploited. It is thus tempting to switch the matrix storage to fp16; however, special care has to be taken not to lose the accuracy boost of tensor cores in doing so. Indeed, if one simply switches the matrix A to fp16 in Algorithm 7.2, the update operations $A_{ij} \leftarrow A_{ij} - L_{ik}U_{kj}$ will accumulate in fp16 arithmetic even when using tensor cores with fp32 accumulation. This will negate almost entirely any accuracy benefit from using tensor cores. In **[27]** we propose a solution to this issue which allows for storing the matrix in fp16 while retaining the accuracy boost of tensor cores. The key idea is to use a left-looking algorithm (Algorithm 7.1) instead of a right-looking one (Algorithm 7.2). Indeed, in the left-looking algorithm, all updates to a given block A_{ik} are performed together one after the other. This allows for introducing a temporary fp32 buffer B_{ik} in which we accumulate the updates in fp32 arithmetic: we change the operation

$$A_{ik} \leftarrow A_{ik} - \sum_{j=1}^{k-1} L_{ij}U_{jk}$$

in Algorithm 7.1 to

$$\begin{aligned} B_{ik} &= \text{fp32}(A_{ik}) \\ B_{ik} &\leftarrow B_{ik} - \sum_{j=1}^{k-1} L_{ij}U_{jk}, \\ A_{ik} &= \text{fp16}(B_{ik}). \end{aligned}$$

The updates to the blocks A_{ki} are handled similarly. We carry out a rounding error analysis that proves that this modification preserves an error bound where the term proportional to u_{low} is independent of n. Since this approach only requires fp32 buffers of small size, it halves the memory footprint of algorithm. Moreover, we also analyze the volume of data movement and show that it is also halved. As a result, the factorization is significantly faster: on A100 GPUs, we obtain a peak performance of 170 TFLOPS, whereas the algorithm of Haidar et al. using fp32 storage only achieves 52 TFLOPS (see Figure 6 in **[27]**).

We also analyze in **[27]** the effect of the precision used for the FACTOR step. We show that rather than performing this step entirely in precision u_{low} or u_{high}, the best performance–accuracy tradeoff is achieved by doubly partitioning the matrix and using the mixed precision tensor cores in the FACTOR step as well.

10.1.3 LU-IR with sparse direct solvers

Sparsity presents both opportunities and obstacles to the use of IR.

On the one hand, while LU factorization of dense matrices in single precision tends to run twice as fast as in double precision, this speedup may not be attained for sparse matrices. This is notably because the analysis phase of the solver (matrix reordering and symbolic factorization) does not involve floating-point arithmetic and so does not benefit from reducing the precision. More subtly, the ratio between the cost of the LU factorization and the LU substitutions in the refinement loop is typically smaller for sparse systems. Indeed, while this ratio is of order $n^3/n^2 = n$ for dense systems, it is only of order $n^2/n^{4/3} = n^{2/3}$ for sparse systems arising from a regular 3D problem, and this ratio is even smaller for a 2D problem (see Section 7.2 and Table 9.1). Therefore, the relative cost of the LU factorization is lower for sparse systems, which provides less room to amortize the cost of the iterations in the refinement loop.

On the other hand, some of the features of iterative refinement are especially attractive when the matrix is sparse. First, as we explain in **[15]**, iterative refinement with a lower precision LU factorization can lead to significant memory savings due to the fact that the LU factors of a sparse matrix are typically much denser, and unlike for dense matrices, the overhead of keeping a high precision copy of the original matrix is negligible. Second, as mentioned in Section 7.3, to best preserve the sparsity of the matrix, sparse direct solvers often employ relaxed pivoting strategies, such as threshold partial pivoting [118, Chap. 7] or the more aggressive static pivoting [186], which can lead to large growth factors; iterative refinement can overcome the resulting numerical instability.

In **[15]**, we implement LU-IR with the sparse direct solver MUMPS, and compare it to GMRES-IR as discussed in the next section.

10.2 GMRES-based iterative refinement (GMRES-IR)

As evidenced by Theorem 10.1, one of the main limitations of LU-IR is that its success is guaranteed only when $\||A^{-1}||\widehat{L}||\widehat{U}|\| \ll 1$, which reduces to $\kappa(A)u_f \ll 1$ for a stable LU factorization. Thus, if the LU factorization is computed in low precision, LU-IR is limited to well-conditioned matrices. This limitation can be overcome by using a more accurate solver for the correction system $Ad = r$. In particular, a natural idea is to solve the system by GMRES (see Algorithm 8.1), using the low precision LU factors as a preconditioner. This GMRES-IR approach is presented in Algorithm 10.2, in which GMRES is used to solve the left-preconditioned system $\widetilde{A}d = U^{-1}L^{-1}Pr$, where $\widetilde{A} = U^{-1}L^{-1}PA$. The products with $\widetilde{A}$ are computed in a precision u_p that is potentially different from the precision u_g of the rest of the GMRES operations, for reasons that will be made clear shortly.

Algorithm 10.2 GMRES-IR with LU preconditioner.

Input: A, b.
Output: An approximate solution to $Ax = b$.

1: Compute the factorization $PA = LU$ in precision u_f.
2: Solve $LUx_0 = Pb$ by substitution in precision u_f.
3: **for** $i = 0$ **to** $i_{\max}$ or until converged **do**
4: Compute $r_i = b - Ax_i$ in precision u_r.
5: Solve $U^{-1}L^{-1}PAd_i = U^{-1}L^{-1}Pr_i$ GMRES in precision u_g with the products with $U^{-1}L^{-1}PA$ in precision u_p.
6: Update $x_{i+1} = x_i + d_i$ in precision u.
7: **end for**

GMRES-IR was proposed by Carson and Higham [87] and was originally intended to solve linear systems nearly singular to the working precision u; however, Carson and Higham [88] subsequently proposed using it to exploit LU factors computed in a lower precision. The rationale in this case is that while solving the correction equations $Ad_i = r_i$ by direct substitution can only provide a relative accuracy of order $\kappa(A)u_f$, GMRES does not share the same limitation. In particular, Carson and Higham [88] analyze Algorithm 10.2 with the constraints $u_g = u$ and $u_p = u^2$. With a stable GMRES implementation such as MGS-GMRES (Section 8.2, [219]), the correction equations are solved with relative accuracy $\kappa(\widetilde{A})u$, which is a significant improvement since we have $u \ll u_f$ and usually also $\kappa(\widetilde{A}) \ll \kappa(A)$. However, the constraints $u_g = u$ and especially $u_p = u^2$ are quite strong in practice. Notably they require the LU factors to be applied in precision u^2, *twice* the target precision (hence, if we target double precision accuracy, we would need to apply them in quadruple precision, which is very slow on most systems).

In **[11]**, we relax these requirements by allowing the products with $\widetilde{A}$ to be performed in a precision u_p possibly lower than u^2, and the rest of the GMRES computations to be performed in a precision u_g possibly lower than u. This results in the five-precision algorithm described in Algorithm 10.2. Analyzing this algorithm requires extending the stability analysis of Paige et al. [219] to the two-precision preconditioned GMRES used here; as discussed in Section 8.2 (see Theorem 8.2), we obtain a backward error bound of order $\kappa(A)u_p + u_g$. The correction equations are thus solved with relative accuracy $\kappa(\widetilde{A})(\kappa(A)u_p + u_g)$, and we obtain the following theorem.

Theorem 10.4 (Section 3.3 of **[11]**). *Let GMRES-IR with LU preconditioner (Algorithm 10.2) be applied to a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is nonsingular. If $\phi = \kappa(\widetilde{A})(u_g + \kappa(A)u_p) \le \kappa(A)^2 u_f^2(u_g + \kappa(A)u_p)$ is sufficiently less than 1 then at each refinement step the forward error is reduced by a factor ϕ until an iterate $\widehat{x}$ is produced for which*

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \le 4pu_r \operatorname{cond}_\infty(A, x) + u, \tag{10.4}$$

where p is the maximum number of nonzeros in any row of $[A\ b]$.

Theorem 10.4 shows that GMRES-IR and LU-IR can attain the same limiting accuracy (10.4), which depends on u and u_r. GMRES-IR improves the convergence condition from $\kappa(A)u_f \ll 1$ to $\kappa(\widetilde{A})(\kappa(A)u_p + u_g) \ll 1$, which depends on u_p, u_g, and the condition number $\kappa(\widetilde{A})$; since $\widetilde{A}$ is preconditioned by the LU factors computed in precision u_f, the convergence of GMRES-IR also depends on u_f through the bound $\kappa(\widetilde{A}) \le \kappa(A)^2 u_f^2$ [88, Eq. (8.3)]. Note that this bound is often pessimistic and that we typically have $\kappa(\widetilde{A}) \approx \kappa(A)u_f$.

Algorithm 10.2 allows for a flexible tradeoff between performance and robustness depending on the choice of its five precisions. There are thousands of possible combinations, which we narrow down in **[11]** to a few combinations of practical interest. One combination of particular interest uses low precision only for the LU factorization ($u_f = u_{\text{low}}$) and double precision for the rest of the operations ($u_g = u_p = u_r = u = u_{\text{high}}$). Haidar et al. [153, 152] implement this variant with GPUs using fp16 as the low precision and fp64 as the high precision; they show that for several matrices for which LU-IR takes a large number of iterations to converge, GMRES-IR converges in a much smaller number of iterations and, for some of them, retains an attractive performance boost compared with LU-IR with an fp32 factorization. However, a fast convergence can be obtained with LU-IR by using the mixed precision fp16/fp32 tensor cores units, as discussed in Section 10.1.2.

In **[15]** we implement the same combination but with fp32 as the low precision, using the multifrontal solver MUMPS for the LU factorization, and we compare this variant of GMRES-IR with LU-IR for solving a range of large and ill-conditioned sparse systems coming from a variety of applications. We obtain reductions of up to a factor 2 in both execution time and memory consumption over the double precision MUMPS solver, with LU-IR being usually faster than GMRES-IR, although the latter is more robust and successfully converges for all test problems.

Note that the bounds on $\kappa(A)$ for convergence guarantees are not always sharp, so it can be difficult to decide which variant should be preferred for a particular problem. To address this issue, Oktay and Carson [209] propose a multistage iterative refinement that switches to increasingly robust but also more expensive variants, starting with LU-IR and moving to a subset of the five-precision GMRES-IR variants analyzed in **[11]**. The decision of when to switch is based on the error bounds obtained by Demmel et al. [109].

10.2.1 GMRES-IR with BLR factorization

As discussed in Chapter 9, in many applications the matrix is amenable to block low-rank (BLR) compression. As established by Theorem 9.1, the stability of the BLR LU factorization is controlled by the low-rank truncation threshold ε. Thus, BLR solvers can be used either as direct solvers (setting ε to the target accuracy) or as preconditioners to iterative methods. In particular, they can be used in conjunction with iterative refinement, both LU-IR or GMRES-IR.

In the study **[15]** already mentioned above, we also investigate the use of BLR compression. This first requires to extend the error analysis of IR to a more general model of approximate LU factorization where the approximations are not necessarily coming from the use of lower precision arithmetic. Theorem 4.1 of **[15]** achieves this, and shows that LU-IR and GMRES-IR converge under the same conditions as usual by replacing u_f (the precision of the factorization) with ε. Since the study focuses on sparse direct solvers, for which relaxed pivoting strategies are quite common, we also investigate the role of the growth factor ρ_n in the convergence of LU-IR and GMRES-IR. Interestingly, we prove that ρ_n multiplies the terms proportional to both u_f and u_p, but not those proportional to ε. Since we usually have $\varepsilon \gg u_f \geq u_p$, this implies that the effect of a large growth factor ρ_n is much less visible when employing BLR compression.

We illustrate experimentally the behavior of IR with BLR compression by using the MUMPS solver. Our results show that IR can still converge with a relatively aggressive compression corresponding to large values of ε, while allowing for large performance gains with respect to the double precision solver: we obtain reductions of the execution time of up to 5.5× and of the memory consumption of up to 3.5×. Moreover, GMRES-IR can converge for larger values of ε than LU-IR, which leads to increased performance in some cases, especially in terms of memory consumption.

10.2.2 GMRES-IR with cheaper (or no) preconditioner

Algorithm 10.3 GMRES-IR with no preconditioner.

Input: A, b, x_0.
Output: An approximate solution to $Ax = b$.

1: **for** $i = 0$ **to** $i_{\max}$ or until converged **do**
2: Compute $r_i = b - Ax_i$ in precision u_r
3: Solve $Ad_i = r_i$ by GMRES in precision u_g.
4: Update $x_{i+1} = x_i + d_i$ in precision u.
5: **end for**

Computing an LU factorization can be expensive, especially for large, sparse matrices. GMRES-IR can also be effective with a cheaper preconditioner M^{-1}, or with no preconditioner at all. In this latter case, Algorithm 10.2 reduces to Algorithm 10.3, which has the form of an inner–outer scheme: the outer loop for iterative refinement (in precision u, with the residual computed in a possibly higher precision u_r), and the inner loop for solving the correction equations with GMRES in lower precision u_g. Assuming a backward stable form of GMRES is used, the correction equations are solved with relative accuracy $\kappa(A)u_g$ and thus we obtain the following theorem.

Theorem 10.5. *Let GMRES-IR with no preconditioner (Algorithm 10.3) be applied to a linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is nonsingular. If $\phi = \kappa(A)u_g$ is sufficiently less than 1 then at each refinement step the forward error is reduced by a factor ϕ until an iterate $\widehat{x}$ is produced for which*

$$\frac{\|x - \widehat{x}\|_\infty}{\|x\|_\infty} \leq 4pu_r \operatorname{cond}_\infty(A, x) + u, \tag{10.5}$$

where p is the maximum number of nonzeros in any row of $[A\ b]$.

Proof. A backward stable form of GMRES in precision u_g will eventually converge to a relative accuracy of order $\kappa(A)u_g$, so this is a direct consequence of Theorem 6.2 of **[10]**. See also Theorem 6.3 of **[12]**. □

Note that the inner GMRES solver can be terminated before reaching a backward error u_g: then in the convergence condition $\kappa(A)u_g \ll 1$ in Theorem 10.5, u_g should be replaced with the backward error actually achieved.

Algorithm 10.3 is one form of mixed precision restarted GMRES; however, to guarantee convergence, the inner GMRES solver must be allowed to continue to iterate until a sufficiently small residual has been achieved, rather than terminating after a fixed number m of iterations, as is more commonly implemented due to cost and memory constraints. However, this result can also provide insight on this more practical GMRES(m) variant. Indeed, even though the convergence of GMRES(m) cannot be guaranteed when $m < n$ (to any level of accuracy, and regardless of the choice of precisions), this theorem does indicate that *if* a relative accuracy $\phi \ll 1$ is reached for sufficiently many outer iterations (cycles), then the process will eventually converge. Moreover, in some cases running GMRES in low precision u_g does not affect its convergence history until the residual becomes close to u_g; in these cases, we can state that *if* the full precision ($u_g = u$) GMRES(m) converges, then its mixed precision form ($u \ll u_g$) will do so too. Naturally, if m inner iterations achieve a residual smaller than u_g, then the use of mixed precision will deteriorate the convergence. Hence, the use of mixed precision tends to be more noticeable for large values of m; this effect is analyzed in detail by Zhao et al. [259].

Algorithm 10.3 was first described by Turner and Walker [241], who perform the inner loop in single precision (u_g) and the outer loop in double precision (u and u_r), and use a fixed number of inner GMRES iterations. Buttari et al. [83] implement several inner–outer iterative algorithms similar to GMRES-IR employing single and double precisions for the solution of sparse linear systems. In particular, one version uses GMRES for the inner loop and FGMRES for the outer loop; this version is also studied by Baboulin et al. [70]. More recent implementations of these methods, still using only single and double precisions, are described by Lindquist et al. [188, 190] and Zhao et al. [259] for CPUs and by Loe et al. [193] for GPUs. Iwashita et al. [170] propose a restarted GMRES where the inner loop uses integer arithmetic and the outer loop uses floating-point arithmetic.

10.2.3 GMRES-IR with mixed precision preconditioned GMRES

The advances in the error analysis of GMRES-IR have been achieved concurrently (and often in the same work) as those in the analysis of GMRES itself. Indeed, the two are closely related since the bounds on the attainable error of GMRES directly determine the convergence conditions (and speed) of GMRES-IR.

The GMRES-IR convergence bounds given in the previous sections predate the most recent and general analysis of mixed precision preconditioned GMRES presented in Section 8.3.1. We have indeed made the choice to keep the above GMRES-IR results in their original form, for consistency with the corresponding papers. However, the more general bounds of Section 8.3.1 are highly relevant in the context of GMRES-IR. Notably, we have seen that it is meaningful to decouple the precision u_p of the products with the preconditioned matrix into two distinct precisions u_m and u_a used for the products with the preconditioner and the original matrix, respectively. In addition, GMRES-IR has focused on using left-preconditioned GMRES for the correction system [87, 88], **[11, 15]**, but we have seen that right and flexible preconditioning can be more (or less) robust depending on the combination of precisions. This motivates a unified experimental study to compare the performance of these different options in the context of GMRES-IR; see Research Problem 16.9.

10.3 BiCGStab-based iterative refinement

GMRES-IR is a general inner–outer method and many of the results described above also apply if the inner GMRES is replaced by another iterative solver. In particular, in this section we discuss the use of BiCGStab (see Section 8.4) and in the next one that of the conjugate gradient (CG) method.

In **[47]**, we investigate BiCGStab-based iterative refinement (hereinafter BiCGStab-IR), that is, iterative refinement with a low precision BiCGStab as inner solver. This amounts to using a restarted BiCGStab solver, which is much less common than restarted GMRES, primarily because restarting GMRES is helpful to contain the size of the Krylov basis, whereas restarting BiCGStab (in a uniform precision context) is generally unhelpful and can on the contrary delay convergence. However, in a mixed precision context, restarting BiCGStab becomes relevant since it allows for attaining high accuracy despite performing most of the operations in low precision. The downside of this approach is that restarting can slow down the convergence; hence there is a tradeoff between the cost of the iterations and their number. We note that the same idea has been investigated by Zhao et al. [258], who also observed the same issue with delayed convergence.

Motivated by this delayed convergence issue, in **[47]** we investigate a variant of BiCGStab called "flying restart" (hereinafter BiCGStab-FR), which was proposed in 1995 by Sleijpen and Van der Vorst [236]. The original motivation for this variant was to improve the attainable

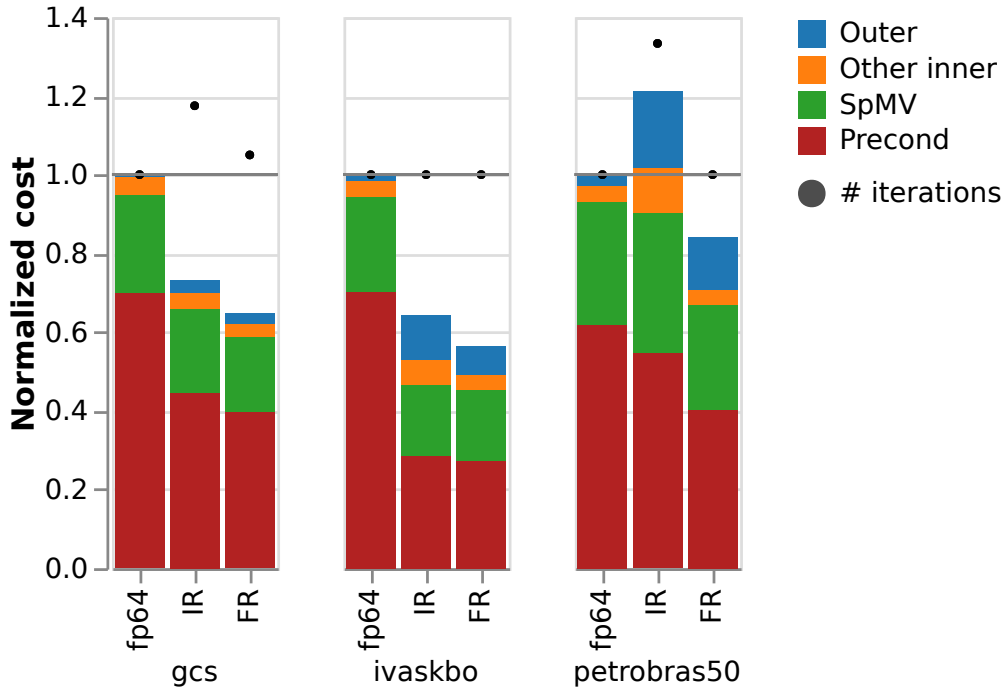


Figure 10.1: Normalized execution time of three solvers (uniform fp64 BiCGStab, BiCGStab-IR, and BiCGStab-FR) for three IFPEN matrices. We indicate the time breakdown in the outer loop and in the inner loop (itsef composed of the SpMVs, the preconditioner application, and other vector operations).

accuracy of BiCGStab in finite (uniform) precision. Similarly to BiCGStab-IR, BiCGStab-FR regularly restarts the solver by replacing the right-hand side with the explicit residual of the current solution; however, BiCGStab-FR does not reset the internal state of the solver and thus can decrease the risk of delaying the convergence. In **[47]**, we propose a mixed precision version of BiCGStab-FR. We show that it also attains high accuracy and can sometimes converge faster than BiCGStab-IR. Moreover, the convergence of this mixed precision BiCGStab-FR method is less sensitive to the restart parameter than BiCGStab-IR, which makes its numerical behavior more consistent.

Figure 10.1 illustrates the performance of mixed precision BiCGStab-IR and BiCGStab-FR, using fp32 for the inner solver and fp64 for the outer loop. The bars correspond to the execution time normalized with respect to the uniform fp64 BiCGStab, and also show a breakdown of the time spent in the outer loop (which initializes the system) and in the inner loop (itsef composed of the SpMVs, the preconditioner application, and other vector operations). This breakdown shows that the inner operations are accelerated thanks to the use of the lower fp32 precision in BiCGStab-IR and BiCGStab-FR. Speedups can thus be expected when this acceleration is achieved without increasing the number of iterations too much. The black dots in the figure show that this is always the case for BiCGStab-FR, which is able to maintain a fast convergence even when BiCGStab-IR does not. Overall, significant speedups of up to nearly the ideal 2× are achieved.

10.4 CG-based iterative refinement

If the matrix A is symmetric positive definite, it is natural to seek to exploit this property to reduce the cost of the solution. Higham and Pranesh [163] investigate a version of Algorithm 10.2 in which GMRES is replaced by CG and the LU factorization is replaced by a Cholesky one.

This raises two potential issues. First, unlike GMRES, CG is not backward stable, and the use of finite precision arithmetic could in principle prevent its convergence. However, the experiments in [163] suggest that in practice, CG often converges as fast as GMRES. Second, the use of low precision may lead to a loss of positive definiteness, which causes the Cholesky factorization to break down. To minimize the risk of this happening, Higham and Pranesh [163] propose to scale and shift the matrix as $A_\ell = \mu(D^{-1}AD^{-1} + cu_f I)$. The two-sided scaling $H = D^{-1}AD^{-1}$, where $D = \mathrm{diag}(a_{ii}^{1/2})$, produces a unit diagonal matrix with off-diagonal elements bounded in magnitude by 1. The diagonal of H is then shifted by cu_f, an amount intended to lift the smallest eigenvalue sufficiently above zero. Finally, a multiplicative factor $\mu = \theta f_{\max}/(1+cu_f)$ is applied, where $\theta < 1$ and $f_{\max}$ is the maximum representable floating-point number in precision u_f. This scaling aims at utilizing as much as possible the range of the low precision, an idea first proposed by Higham, Pranesh and Zounon [164] for LU factorization. As explained in [163], shifting H by a multiple of I is better than shifting A by a multiple of I, as it is equivalent to shifting A by a multiple of $\mathrm{diag}(a_{ii})$ and so it makes the same relative perturbation to each diagonal element of A.

Chapter 11

Adaptive precision algorithms

Summation exhibits a very special property: summing small numbers to large numbers tends to leave the large numbers mostly unchanged. In the most extreme case, the small number may be completely absorbed, lost, and if this occurs repeatedly we run into the phenomenon that we have called "stagnation" (see Section 3.1), which can be problematic. However, more generally, when adding two numbers $a + b$ where $|b| \ll |a|$, the small number b might change the large number a, but only by a small quantity. This is illustrated in Figure 11.1, where many of the least significant bits of b are ignored.

In cases where the sum is ill conditioned and suffers from cancellation, this observation can be quite problematic, since the lost bits of the small number b might be important to the final result. This difficult case requires the use of more accurate summation algorithms, such as those discussed previously in Section 4.3. However, if the sum is well conditioned, or the amount of cancellation is in any case acceptable, then this observation stops being a problem and becomes an opportunity. Indeed, if we accept the fact that most of the bits of b are going to be lost anyway, then we may decide to store b in low precision (with as few as p bits in Figure 11.1), without impacting significantly the accuracy. Moreover, if b is the result of a previous computation, this computation can be performed in low precision.

This key idea is at the foundation of an emerging subclass of mixed precision algorithms, that we have called "adaptive precision" (see section 14 of **[10]**). Adaptive precision algorithms dynamically adapt the precision of each operation or variable based on the data at hand. The precisions are chosen based on the magnitude of the variables being summed together. Since these magnitudes can usually be expected to vary in a somewhat continuous way, adaptive precision algorithms can deliver a similarly continuous level of accuracies. They have thus been called "variable accuracy", as opposed to "variable precision", algorithms. For the same reason, adaptive precision algorithms can leverage a continuum of different precision formats—the more formats are available, the larger gains can be expected, since we can adapt more finely the precisions to the data. Throughout this chapter, we will denote as $\varepsilon > 0$ the desired accuracy

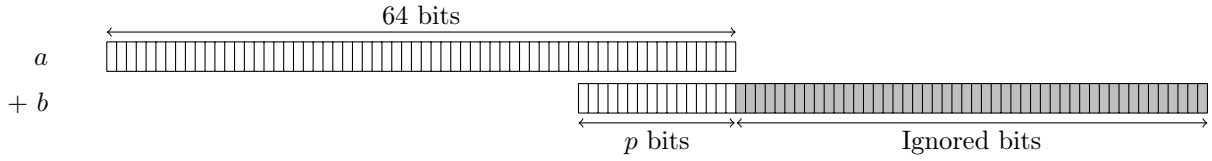


Figure 11.1: Summing a small number to a large number leaves the large one mostly unchanged.

and we will consider p different precisions with unit roundoffs $u_1 \leq \varepsilon < u_2 < \ldots < u_p$ (we only need the first precision to provide an accuracy of at least ε, and the remaining $p-1$ precisions are the "low" precisions).

The contributions described in this chapter are based on the following papers: **[18]**, **[36]** for Section 11.1; **[16]** for Section 11.2.

11.1 Adaptive precision sparse matrix–vector product

In **[18]**, we develop an adaptive precision algorithm for computing a sparse matrix–vector product (SpMV) $y = Ax$. Given p available precisions $u_1,\ldots,u_p$, the idea is to split A into p matrices $A^{(1)}, \ldots, A^{(p)}$ with disjoint sparsity patterns (that is, each element of A belongs to exactly one of the $A^{(k)}$ matrices) and to store matrix $A^{(k)}$ in precision u_k. Then, the SpMV can be efficiently computed as

$$y = Ax = \sum_{k=1}^{p} A^{(k)}x$$

where the partial SpMV $A^{(k)}x$ is performed in precision u_k.

The following result provides a precise criterion to decide which elements of A go into each $A^{(k)}$ matrix, so as to obtain a backward error bounded by a user-controlled parameter ε.

Theorem 11.1 (Theorem 3.1 of **[18]**). *Let $A \in \mathbb{R}^{m\times n}$ and $x \in \mathbb{R}^n$. Let $\varepsilon > 0$, consider p precision parameters such that $u_1 \leq \varepsilon < u_2 < \ldots < u_p$, and let $u_{p+1} = 1$. Define the split $A \approx \sum_{k=1}^{p} A^{(k)}$ such that for any nonzero element a_{ij} we have, for $k = 1\colon p$,*

$$a_{ij}^{(k)} = \begin{cases} \mathrm{fl}_k(a_{ij}) & \text{if } |a_{ij}| \in (\frac{\varepsilon\|A\|}{u_{k+1}}, \frac{\varepsilon\|A\|}{u_k}], \\ 0 & \text{otherwise,} \end{cases} \tag{11.1}$$

where $\mathrm{fl}_k(\cdot)$ denotes the operator that rounds to precision u_k. Let $y^{(k)} = A^{(k)}x$ be evaluated in precision u_k and let $y = \sum_{k=1}^{p} y^{(k)}$ be evaluated in precision u_1. Then the computed $\widehat{y}$ satisfies

$$\widehat{y} = (A + \Delta A)x, \quad \|\Delta A\| \leq c\varepsilon\|A\|, \tag{11.2}$$

where c is a small constant ($c \lesssim q^2$, where q is the maximum number of nonzero elements per row of A).

The criterion (11.1) shows that a given element a_{ij} should be stored in a precision u_k that is inversely proportional to its magnitude $|a_{ij}|$. The bound (11.2) should be compared to the bound $\|\Delta A\| \leq qu_1\|A\|$ obtained by computing the SpMV entirely in uniform precision u_1. By setting $\varepsilon = u_1$ we recover almost the same accuracy while potentially allowing the use of lower precisions if the elements of A exhibit wide variations in magnitude. Moreover, we can also use larger values of ε to achieve any level of desired accuracy that is not constrained by the precision parameters of the available arithmetics. Finally, while (11.2) is a normwise bound, a componentwise one can be obtained by replacing $\|A\|$ with $(|A||x|)_i$ in (11.1) (see Theorem 3.1 and Section 3.1 of **[18]** for details).

Note that, since we have set $u_{p+1} = 1$ in Theorem 11.1, elements of magnitude smaller than $\varepsilon\|A\|$ will not belong to any $A^{(k)}$ and will thus be dropped, that is, replaced with zero. This observation can significantly increase the sparsity of the adaptive matrix representation.

Since the SpMV is a memory-bound operation, its cost is expected to follow the cost of storing A. In order to measure the latter, we need to account for the cost of storing the indices. For

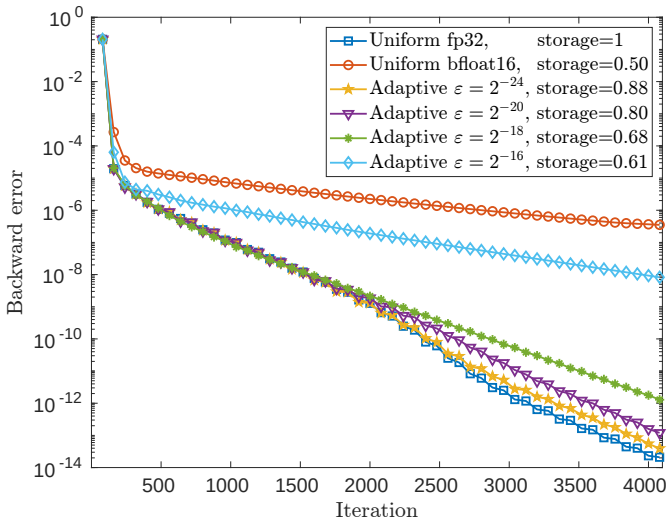


Figure 11.2: Convergence of restarted GMRES for uniform and adaptive precision variants of SpMV. The matrix is ML_Laplace from SuiteSparse, restart size is 80, a diagonal Jacobi preconditioner is used. Storage costs for the matrix (normalized with respect to the uniform fp32 variant) are provided.

example, for a compressed sparse row (CSR) representation, storing A requires $\mathrm{nnz}(A)$ floating-point values and $\mathrm{nnz}(A)+n+1$ indices, where $\mathrm{nnz}(A)$ denotes the number of nonzero elements of A. In bytes, the storage cost is therefore $\mathrm{nnz}(A)f_1 + (\mathrm{nnz}(A)+n+1)i$ where f_1 and i are the byte sizes of floating-point numbers in precision u_1 and of integers, respectively. In comparison, the cost of storing $A^{(1)}, \ldots, A^{(p)}$ is

$$\sum_{k=1}^{p} \Big(\mathrm{nnz}(A^{(k)})f_k + \big(\mathrm{nnz}(A^{(k)})+n+1\big)i \Big) \leq \big(\mathrm{nnz}(A)+p(n+1)\big)i + \sum_{k=1}^{p} \mathrm{nnz}(A^{(k)})f_k, \quad (11.3)$$

where $\sum_{k=1}^{p} \mathrm{nnz}(A^{(k)}) \leq \mathrm{nnz}(A)$ since the $A^{(k)}$ have disjoint sparsity patterns (the left-hand side may be smaller than $\mathrm{nnz}(A)$ if some elements are dropped). Formula (11.3) shows that the adaptive precision representation of A requires at most only $(p-1)(n+1)$ extra indices, an overhead cost which should largely be compensated by the reduction of the size of the floating-point values, provided that the matrix presents sufficient elements that can be switched to low precision ($\mathrm{nnz}(A^{(1)}) \ll \mathrm{nnz}(A)$) and sufficient nonzero elements per row ($\mathrm{nnz}(A) \gg n$).

We carry out an extensive experimental study in **[18]** that confirms that this adaptive precision approach can lead to significant storage reductions for a wide range of sparse matrices from various applications. We also confirm that these storage reductions translate to significant speedups on multicore CPU architectures when using two precisions, fp64 and fp32, which are supported in hardware. In principle, the approach may benefit from a larger number of precisions, and since SpMV is memory-bound, it could be further accelerated by using custom precisions that are not supported in hardware. We undertake this goal in **[36]**, by developing a seven-precision SpMV which, in addition to the standard fp64 and fp32 arithmetics, also uses custom precision formats using 56, 48, 40, 24, and 16 bits (the CPU hardware targeted here does not support half precision arithmetic). Achieving efficiency with these custom formats requires developing the memory accessor that will be described in Section 12.2.

In **[18]**, we also investigate the use of this adaptive precision SpMV to accelerate mixed precision restarted Krylov solvers, including GMRES and BiCGStab. We show that the use of adaptive precision SpMV is particularly attractive in this context, due to several factors. First, if

we use small restart sizes and a cheap preconditioner (or none at all), the SpMV is the bottleneck of the computation. Second, we require many SpMVs where the matrix A is fixed and only the vector changes: thus, the overhead of computing an adaptive precision decomposition of A is negligible since it can be reused for all iterations. Third, as we have explained in Chapter 10, restarted Krylov solvers are a form of iterative refinement that allows for converging to high accuracy even when the inner solvers use lower precision. As illustrated in Figure 11.2, this is doubly beneficial for the adaptive precision SpMV because, first, we can set ε to a larger value which leads to greater storage reductions and, second, we can tune ε to find the best tradeoff between the convergence speed and the cost of the SpMV (remember that ε is a continuous parameter that need not match the unit roundoff of any specific floating-point arithmetic).

11.2 Adaptive precision low-rank approximation

In Section 6.3, we have described an adaptive precision low-rank approximation from **[16]** that approximates a low-rank matrix $\bar{U}\bar{\Sigma}\bar{V}^T$ as

$$\widehat{\bar{U}}\widehat{\bar{\Sigma}}\widehat{\bar{V}}^T = \begin{bmatrix}\widehat{U}_1 \dots \widehat{U}_p\end{bmatrix} \begin{bmatrix} \widehat{\Sigma}_1 & & \\ & \ddots & \\ & & \widehat{\Sigma}_p \end{bmatrix} \begin{bmatrix}\widehat{V}_1 \dots \widehat{V}_p\end{bmatrix}^T \tag{11.4}$$

where $\widehat{U}_k$, $\widehat{\Sigma}_k$, and $\widehat{V}_k$ are stored precision u_k, and we have shown in Theorem 6.1 that the partitioning (11.4) should be built such that $\|\Sigma_k\| \leq \varepsilon\|A\|/u_k$. This criterion shows that the singular values and their associated singular vectors can be stored in a precision that is inversely proportional to their magnitude. It is similar to the criterion on the magnitudes of the matrix elements for the adaptive precision SpMV presented in the previous Section 11.1. In fact, the two methods share a link: indeed, (11.4) can be rewritten as a sum of p matrices, $\widehat{\bar{U}}\widehat{\bar{\Sigma}}\widehat{\bar{V}}^T = \sum_{k=1}^{p} \widehat{U}_k\widehat{\Sigma}_k\widehat{V}_k^T$, where the coefficients of $\widehat{U}_k\widehat{\Sigma}_k\widehat{V}_k^T$ are bounded in magnitude by $\|\widehat{\Sigma}_k\|$. Thus the SpMV criterion $|\widehat{U}_k\widehat{\Sigma}_k\widehat{V}_k^T|_{ij} \leq \varepsilon\|A\|/u_k$ is consistent with the LRA criterion $\|\Sigma_k\| \leq \epsilon\|A\|/u_k$.

Like other adaptive precision methods, the choice of precisions is adapted to the data at hand, and so the potential of this approach is completely dependent on the matrix that it is used on. In this case, significant gains can be expected if the singular values of the matrix decay rapidly.

11.3 Adaptive precision block Jacobi preconditioner

A popular preconditioner for iterative methods is the block Jacobi method, which uses the block diagonal matrix

$$M = D = \begin{bmatrix} D_1 & & \\ & \ddots & \\ & & D_q \end{bmatrix}$$

as preconditioner. D^{-1} can be efficiently applied by inverting each of its blocks D_i independently:

$$y = D^{-1}x = \begin{bmatrix} D_1^{-1}x_1 \\ \vdots \\ D_q^{-1}x_q \end{bmatrix}.$$

Anzt et al. [65] propose an adaptive precision preconditioner which uses different precisions for different blocks D_i. This is motivated by the fact that, in floating-point arithmetic, the computed

$\widehat{y}$ satisfies

$$\widehat{y} = D^{-1}x + \Delta y = \begin{bmatrix} D_1^{-1}x_1 + \Delta y_1 \\ \vdots \\ D_q^{-1}x_q + \Delta y_q \end{bmatrix}, \quad \|\Delta y_i\| \leq c_i \varepsilon_i \kappa(D_i),$$

where ε_i is the precision used for inverting D_i and c_i is a constant depending on the inversion method (for example, if we use an LU factorization of $D_i \in \mathbb{R}^{n\times n}$, $c_i \approx 3n^3\rho_n$ by Theorem 7.4).

Therefore, if we wish to have $\|\Delta y\| \leq \varepsilon$ for some $\varepsilon > 0$ target accuracy, we should balance each of the error terms $\|\Delta y_i\|$, or rather their bounds, so that $c_i\varepsilon_i\kappa(D_i) \approx \varepsilon$ for all $i = 1\colon q$. If we assume the constants c_i to be of comparable size (their effect on the actual error $\|\Delta y_i\|$ is anyway known to be pessimistic, as discussed in Chapter 3), we obtain the criterion $\varepsilon_i \approx \varepsilon/\kappa(D_i)$. Hence with p different precisions $u_1 \leq \varepsilon < u_2 < \ldots < u_p$ at our disposal, we should set $\varepsilon_i = \max_k \{u_k : u_k \leq \varepsilon/\kappa(D_i)\}$.

This adaptive precision block Jacobi preconditioner has been implemented in the Ginkgo library [129], where it is shown to accelerate the preconditioned CG by up to 30%. The implementation uses up to six precision formats, including custom ones, for storing the explicitly inverted blocks D_i^{-1}, and applies them in double precision, using a memory accessor for performance (see the next Chapter 12).

The approach presented in this section is not exclusive to block Jacobi preconditioners. It can apply to other types of preconditioners that also rely on inverting subparts of a given matrix separately. This is in particular the case for domain decomposition preconditioners, such as the additive Schwarz method: see Research Problem 16.20.

11.4 Adaptive precision relaxed GMRES

The mixed precision relaxed GMRES method described in Section 8.3.3 shows that the precision of the matrix–vector product can be adapted to be inversely proportional to the norm of the residual, which is a continuously decreasing quantity throughout the iterations. Therefore, relaxed GMRES is well suited to an adaptive precision implementation: the more precision formats are available, the more we can benefit from this observation. Moreover, we can combine relaxed GMRES with the adaptive precision SpMV algorithm presented in Section 11.1: this leads to a doubly adaptive method, in the sense that the precisions vary both across matrix coefficients and across iterations; see Research Problem 16.18.

Chapter 12

Memory accessors

This chapter is interested in strategies that decouple the precisions used for storage and for computations. Such strategies are beneficial in various contexts, many of which we have discussed in previous chapters of this manuscript. For example, in the context of preconditioned GMRES (Section 8.3.1) and GMRES-based iterative refinement (Section 10.2), we may want to store the preconditioner in low precision but apply it in high precision. If the preconditioner is an LU factorization of the matrix, this requires performing triangular solves in a compute precision higher than the LU storage precision. Another important application is to handle a storage precision based on custom floating-point types with no hardware support. In this case we must indeed use a higher compute precision based on a standard type with native support.

In order to achieve efficiency, strategies that decouple the storage and compute precisions are based on so-called memory accessors: the idea is to access the data in low precision and convert it on the fly to high precision before performing arithmetic operations on it. This allows for reducing the cost of data transfers and should therefore improve the performance of memory-bound computations. This concept of memory accessor was first proposed by Anzt et al. [66] and implemented in the Ginkgo library [146] to accelerate various memory-bound BLAS kernels such as matrix–vector products and triangular solves.

Memory accessors are a critical tool for the efficiency of several adaptive precision algorithms (Chapter 11), which can naturally exploit a continuum of precision formats for storing the data. For example, memory accessors have been developed by Flegar et al. [129] for the adaptive precision block Jacobi preconditioner (Section 11.3), and by Kriemann [176, 177] for hierarchical matrix–vector products based on adaptive precision low-rank approximations (Sections 6.3 and 11.2). This chapter will present memory accessors developed for the adaptive precision sparse matrix–vector product (Section 11.1) and the adaptive precision block low-rank triangular solve (Section 9.4).

The contributions described in this chapter are based on the following papers: **[41]** for Sections 12.1, 12.3 and 12.4; **[36]** for Section 12.2;

12.1 Efficient conversion from custom to standard floating-point formats

A central component of memory accessors is the conversion from the storage (often custom) precision formats to the compute precision format. This conversion must indeed be efficiently implemented in order be lightweight and leave the performance bottleneck on the memory transfers.

Table 12.1: List of custom reduced precision (rp) formats obtained by truncating the standard fp64 and fp32 formats.

Format	Numbers of bits			Unit roundoff
	Sign	Exponent	Significand	
fp64	1	11	52 (+1)	$2^{-53} \approx 1 \times 10^{-16}$
rp56	1	11	44 (+1)	$2^{-45} \approx 3 \times 10^{-14}$
rp48	1	11	36 (+1)	$2^{-37} \approx 7 \times 10^{-12}$
rp40	1	11	28 (+1)	$2^{-29} \approx 2 \times 10^{-9}$
fp32	1	8	23 (+1)	$2^{-24} \approx 6 \times 10^{-8}$
rp24	1	8	15 (+1)	$2^{-16} \approx 2 \times 10^{-5}$
rp16	1	8	7 (+1)	$2^{-8} \approx 4 \times 10^{-3}$

In this section, we describe the conversion implemented in **[41]** that will be used for Sections 12.3 and 12.4; the one implemented in **[36]** and used in Section 12.2 is very similar.

One simple approach to define custom floating-point formats is to take a standard precision format, such as fp64 or fp32, and truncate some of the least significant bits. Table 12.1 defines five such custom "reduced precision" (rp) formats: rp56, rp48, and rp40 are obtained by truncating the fp64 format, and rp24 and rp16 are obtained by truncating the fp32 format (note that rp16 is in fact the bfloat16 format, although in this chapter we do not assume hardware support for bfloat16 arithmetic).

Note that we have restricted our focus to custom formats with a total number of bits that is a multiple of 8 (that is, an integer number of bytes), because this greatly simplifies their handling. Indeed, we can represent a custom rp number as an array of 8-bit integers (`uint8_t`). For example, the rp48 format can be represented as `uint8_t[6]`. We can then write the conversion in a relative lightweight fashion by only using bit shifts and additions on the standard integer types. Note that, for efficiency, we do not process each of these 6 bytes individually (as `uint8_t`). Rather, we decompose this 6-byte array as the concatenation of a 4-byte array and a 2-byte one, which we can process as `uint32_t` and `uint16_t`, respectively. We thus only need to manipulate two integers, which reduces the number of required bitshifts and additions. Note that the number of integers that are needed depends on the target custom format: for example, for the 56-bit (7-byte) rp56 format, three integers (`uint32_t`, `uint16_t`, and `uint8_t`) are needed.

When converting between the standard fp64 format and the custom formats rp24 and rp16, we also need to handle the different number of bits used for the exponent. In order to do that efficiently, we use the standard fp32 format as intermediary. For example, for converting from rp16 to fp64, we first convert from rp16 to fp32 in a similar fashion as described above, and then convert from fp32 to fp64 with a standard conversion. This illustrates that, if the compute precision is fp64, using custom formats with 8-bit exponent as storage precision requires an extra conversion between fp32 and fp64. The cost of this extra conversion is however small and usually worth gaining 3 bits of precision compared with 11-bit exponent formats that could in principle be obtained by directly truncating the significand of an fp64 number.

So far, we have discussed how to convert a single scalar number, but in many cases we will need to convert an entire array of such numbers. Naturally, we could simply loop on each element and apply the conversion described above. However, a more efficient conversion can be obtained by exploiting vectorized instructions. Specifically, in **[41]**, we show how to use the `permutexvar` instruction, which is available on recent hardware that possess the `AVX512-VBMI` feature flag. The `permutexvar` instruction takes a permutation array [168, volume 2C, pp. 471–472] containing as many elements as there are in the register (for example, to permute a 512-bit register byte-wise,

the permutation array holds 64 elements). We can therefore use different permutation arrays to define different custom formats by specifying which parts of each element to extract (see Figure 5.2 in **[41]** for an illustration). This vectorized instruction allows for converting multiple numbers in a single instruction, regardless of the custom format, and is therefore quite efficient.

12.2 Memory accessor for sparse matrix–vector product

Sparse matrix–vector product (SpMV) is a key kernel in many applications, in particular the solution of sparse linear systems via iterative methods, for which the SpMV can represent a significant fraction of the total cost. On the one hand, storing the matrix in lower precision allows for reducing its storage cost and, since SpMV is a memory-bound operation, its time cost. On the other hand, performing the SpMV in a higher precision can significantly improve the attainable error bound, as we discussed in the case of GMRES in Section 8.3.1. Therefore, a memory accessor for SpMV is of high interest.

An important point to consider when analyzing the performance of SpMV is that the volume of memory accesses depends not only on the matrix values but also on the indices, and thus on the specific sparse matrix format that is used. Since the use of low precision only reduces the storage for the values, but not for the indices, sparse formats that minimize the weight of the latter will make the most out of mixed precision memory accessors. For example, Grützmacher et al. [146] develop a memory accessor for the ELL format [76], which employs padding to force the same number of nonzeros per row, and is particularly suited for GPUs [64]. Mukunoki et al. [203] develop a memory accessor for the standard compressed sparse row (CSR) format and for the adaptive multilevel blocking (AMB) format [206], which can significantly reduce the weight of the indices. They compare the performance of both formats and show that AMB can often achieve larger speedups from the use of low precision than CSR.

As we have presented in Section 11.1, matrices exhibiting values with wide variations in magnitude can significantly benefit from the use of an adaptive precision representation in which gradually smaller elements are stored in gradually lower precisions **[18]**. The more precision formats are available, the more we can adapt the representation to the elements' magnitudes. This motivates developing a memory accessor for adaptive precision SpMV with efficient support for custom precision formats. We do so in **[36]** by extending the accessor of Mukunoki et al. [203]. As explained in Section 11.1, the CSR format allows for a simple and efficient implementation of the adaptive precision SpMV, and therefore we focus on this format; see Research Problem 16.17 regarding the extension to other matrix formats. Our implementation uses the seven custom precision formats defined in Table 12.1 and implements the conversion in a very similar fashion as what is described in Section 12.1.

Figure 12.1 illustrates the performance of this memory accessor–based, seven-precision SpMV for the Long_Coup_dt0 matrix of the SuiteSparse collection. We compare the storage and time costs of the uniform and adaptive precision SpMV variants for various accuracies ε corresponding to the unit roundoffs of the formats. This allows for directly comparing the uniform and adaptive variants for an equivalent accuracy ε, where the uniform variant uses the corresponding precision for all matrix values, whereas the adaptive variant also uses lower precisions (for the small matrix values) and dropping (that is, matrix values less than ε in magnitude are replaced with zero). Both variants use 32-bit integers for the indices. The figure shows that, as expected, the time cost of the SpMV is directly proportional to the storage cost. As mentioned above, the cost of the uniform precision variant comprises an incompressible part corresponding to the 32-bit indices, which explains why, even with 16-bit values, the uniform precision storage is still 50% of the fp64 storage. The adaptive precision variant does not share this limitation because, in addition to lower precisions, it also employs dropping, which increases the sparsity of the matrix and thus

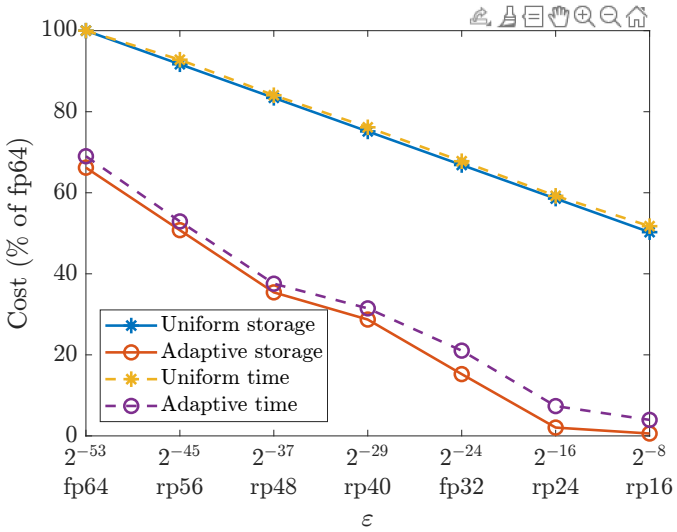


Figure 12.1: Storage and time cost of uniform and adaptive precision SpMV for various accuracies ε (corresponding to the unit roundoffs of the formats defined in Table 12.1).

reduces the storage cost of the indices. Overall, these performance results show that SpMV can be significantly accelerated by reducing the storage precision and confirm the relevance of using custom floating-point formats.

In **[36]**, we also investigate the use of custom formats with reduced exponent. This is motivated by the observation that, with the seven precisions defined in Table 12.1, coefficients being stored in the same precision have magnitudes that can differ by a factor of at most 2^8—the ratio between the unit roundoffs of two consecutive precisions. Hence, only 3 exponent bits are sufficient to represent the coefficients; this represents a significant opportunity to further reduce storage compared with the formats in Table 12.1 that use 11 or 8 exponent bits. We extend the conversion operations described in Section 12.1 to also handle a reduced exponent. This creates a tradeoff because the exponent manipulation makes the conversion more expensive, introducing an overhead cost that we hope to amortize with the reduced storage. Our experiments show that the use of reduced exponents sometimes improves performance and other times degrades it.

12.3 BLAS-based block memory accessor for dense triangular solve

One limitation of most existing memory accessor approaches is that the data is accessed at the scalar (elementwise) level: the matrix coefficients are converted from the storage to the compute precision one at a time. This requires entirely handcoding the target kernel, which makes achieving efficiency and portability difficult, and goes against the traditional separation of concerns whereby high level linear algebra operations rely on low level BLAS kernels optimized for a particular architecture.

In **[41]**, we take a different approach: we propose a BLAS-based block memory accessor that loads into memory and converts entire blocks of scalars at a time, and relies on BLAS libraries to perform computations on these blocks. This approach avoids the heavy programming duty of rewriting and extending the BLAS, and allows for exploiting optimized uniform precision BLAS kernels. This block approach has been much less investigated than the scalar approach; it has also been used in [176], where it is called a "semi-on-the-fly" approach.

We perform a thorough investigation of this block approach focusing on triangular solves. We consider both dense matrices (this section) and matrices compressed with BLR approximations (Section 12.4). Algorithm 12.1 presents a mixed precision implementation for an upper triangular solve $Ux = y$ with two precisions: u_{low}, the low storage precision, and u_{high}, the high compute precision. The matrix U is partitioned into $b \times b$ blocks U_{ij} and the system $Ux = y$ is solved by blocks. Whenever a given block U_{ij} needs to be used, it is loaded from the main memory, where it is stored in precision u_{low}, and converted (upcasted) to precision u_{high} into a temporary workspace B. The computation is then performed with B in precision u_{high}. The workspace B is reused for all blocks U_{ij} and therefore we only require an extra storage of manageable size $b \times b$. All BLAS calls are executed with standard uniform precision routines (`trsv` and `gemv`); such routines are highly optimized for a given target architecture and can thus be expected to be very efficient. Moreover, the conversion operations from precision u_{low} to precision u_{high} can also be efficiently implemented as described in Section 12.1. Therefore, the performance of Algorithm 12.1 should be mainly driven by the data transfers, which are reduced thanks to the low storage precision.

Algorithm 12.1 Mixed precision triangular solve with a block memory accessor.

Input: U, an $m \times m$ upper triangular matrix stored in precision u_{low} and partitioned into $b \times b$ blocks U_{ij}, with $m = pb$; y, an m-vector stored in precision u_{high}; B, a $b \times b$ workspace stored in precision u_{high}.
Output: x, an m-vector stored in precision u_{high}, solution to $Ux = y$.

1: Initialize $x = y$.
2: **for** $j = p, \ldots, 1$ **do**
3: Read U_{jj}, upcast it to precision u_{high}, and store it in B.
4: Solve $Bx_j = x_j$ in precision u_{high} using the BLAS kernel `trsv`.
5: **for** $i = j - 1, \ldots, 1$ **do**
6: Read U_{ij}, upcast it to precision u_{high}, and store it in B.
7: Compute $x_i = x_i - Bx_j$ in precision u_{high} using the BLAS kernel `gemv`.
8: **end for**
9: **end for**

The block size parameter b presents a tradeoff between the efficiency of the BLAS calls and the efficiency of the data transfers. Indeed, in order for the data transfers to be efficient, it is important that the workspace B remains in the fast memory before the BLAS call, since if it spills back into the slow memory, it would need to be reloaded in high precision. Therefore, b must be sufficiently large for the BLAS calls to be efficient, but sufficiently small for B to fit into the fast memory. Note that what constitutes the "fast" and the "slow" memory is architecture dependent; we may expect the registers and highest levels of cache to be fast, and the main memory to be slow.

The performance and behavior of Algorithm 12.1 is illustrated in Figure 12.2. We consider a parallel configuration corresponding to the bottom of the elimination tree in sparse direct methods (see Section 7.2) where each thread solves its own independent triangular system of order $m = 4096$. The compute precision u_{high} is set to fp64 and we compare various lower storage precisions u_{low} corresponding to Table 12.1. We also report the performance of the uniform precision triangular solve in fp64 or fp32 arithmetic for reference. The figure illustrates several of the points mentioned above. First, we can see that small block sizes achieve poor efficiency; performance quickly improves as b increases and as the BLAS calls become more efficient, until it reaches a plateau. For the uniform precision variants, this plateau persists no matter how large b becomes, whereas the behavior of the mixed precision accessor variants is

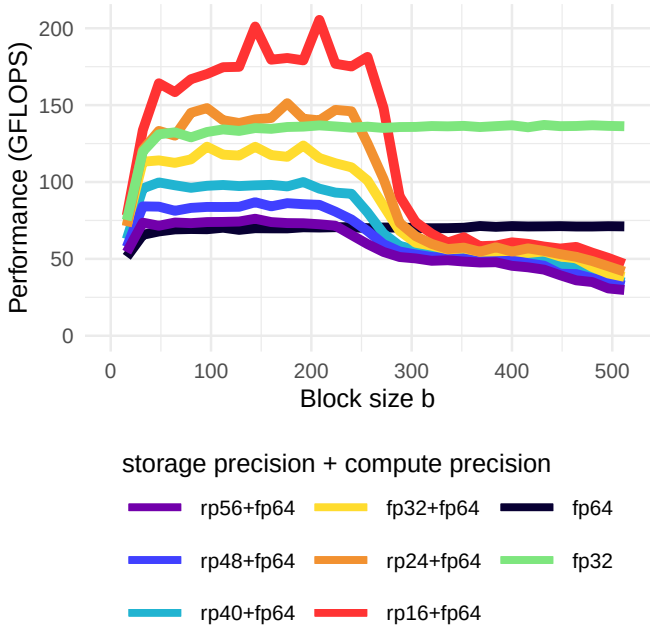


Figure 12.2: Performance of the block memory accessor for triangular solves, with fp64 as the compute precision and various lower storage precisions, depending on the block size b. Uniform fp64 and fp32 precision variants are also provided for reference.

very different, with an abrupt performance drop occurring when b exceeds a certain value. We have performed a detailed analysis in [41] that relates this value to the maximum block size such that two copies of the block (one in precision u_{high} and one in precision u_{low}) still fit into the L1 and L2 caches of the computer; this explains why the performance drop occurs slightly later for lower storage precisions. Most importantly, the performance plateau of each variant is higher for lower storage precisions. Therefore, provided that a suitable block size is chosen, our block memory accessor is able to translate the continuum of storage precisions into a continuum of performance.

12.4 Memory accessor for block low-rank triangular solve

In [41], we also describe how to extend Algorithm 12.1 to block low-rank (BLR) matrices (see Chapter 9). At first sight, this may seem straightforward: a naive implementation would be to use the same block size for the BLR approximation and for the memory accessor and, whenever a low-rank block must be accessed, load its low-rank factors into the fast memory and convert them to high precision. However, the BLR block size, while under our control, must be set to a sufficiently large value to achieve high compression rates. As explained in Section 9.1, this is because the BLR block size should increase with the matrix size m; in practice, even for medium-sized matrices, BLR block sizes of order 500 are typically necessary to obtain good performance [23, 13]. Based on the experiments with the dense triangular solve reported in Figure 12.2, this naive approach would therefore not be efficient since the memory accessor requires much smaller block sizes that fit into the fast memory. However, it is important to note that this difficulty mainly concerns full-rank blocks. Indeed, low-rank blocks of large size may

Table 12.2: Impact of the memory accessor on the adaptive precision BLR triangular solve in MUMPS (Queen_4147 matrix, $\varepsilon = 3 \times 10^{-9}$).

	Compute precision	Storage precisions	Accessor method	Backward error	LU factor storage (GB)	Solve time (ms)
Full-rank	fp64	fp64	—	10^{-16}	112	871
BLR	fp64	fp64	—	10^{-11}	57	505
BLR	fp64	fp64, fp32	Naive	10^{-11}	44	922
BLR	fp64	fp64, fp32	**[41]**	10^{-11}	44	509
BLR	fp64	fp64, rp56, ..., rp16	Naive	10^{-11}	37	1089
BLR	fp64	fp64, rp56, ..., rp16	**[41]**	10^{-11}	37	479

still fit into the fast memory if their rank is small enough. Therefore, one idea to overcome this issue is to further partition the full-rank blocks into smaller subblocks and to load them one subblock at a time. This is essentially a recursive memory accessor: for example, to perform the dense triangular solves with the (full-rank) diagonal blocks, we rely on Algorithm 12.1 where b plays the role of the subblock size and m the role of the BLR block size. Similarly, we use a `gemv` memory accessor for the matrix–vector products with the off-diagonal blocks.

The experiments presented in **[41]** confirm that, provided the BLR compression rate is large enough, subblocking the full-rank blocking significantly extends the range of block sizes for which the memory accessor performance is optimal (see Figure 6.2 in **[41]**).

Moreover, we have also extended our memory accessor to the adaptive precision BLR representation described in Section 9.4, in which the low-rank blocks are further partitioned in multiple lower rank components each stored in a different precision. In this context, efficient support for custom floating-point formats is crucial for performance. We have integrated the ingredients presented in the previous sections (efficient conversion routines, a BLAS-based block memory accessor, subblocking for BLR matrices, etc.) in the MUMPS solver. Table 12.2 illustrates the resulting benefits: with the naive accessor originally implemented in MUMPS, the storage reductions achieved by using adaptive precision did not lead to corresponding time reductions but were on the contrary achieved at the expense of time. With our new accessor from **[41]**, we obtain significant speedups (greater than 2×), which brings the adaptive precision solve back to being as fast as the uniform precision one, or even slightly faster when the full continuum of precisions is used. These results illustrate the potential of memory accessors to optimize the storage cost of the solver while maintaining its efficiency and accuracy.

Chapter 13

Butterfly factorizations

This chapter deals with the problem of butterfly factorization, which amounts to writing an $n \times n$ (usually dense) matrix Z, with $n = 2^L$, as a product $B_1 \dots B_L$, where each butterfly factor B_ℓ is an $n \times n$ sparse matrix with a fixed sparsity pattern. Such a factorization is useful to reduce the time and memory complexity of computations involving matrix Z. Butterfly factorizations appear in particular in fast transforms such as the discrete Fourier transform or the Hadamard transform [101], and, thanks to their strong expressivity and extreme sparsity, find a wide range of applications in various domains such as signal processing, machine learning, and numerical linear algebra.

We first review some basic properties of butterfly factorizations and discuss their possible applications. Then, we describe the main contribution of this chapter, which deals with the quantization of butterfly factorizations in floating-point arithmetic. The presented methods are based on the optimal quantization of rank-one matrices described in Section 6.7.

The contributions described in this chapter are based on the following papers: **[42]** for Section 13.3.

13.1 Butterfly factorizations via rank-one approximations

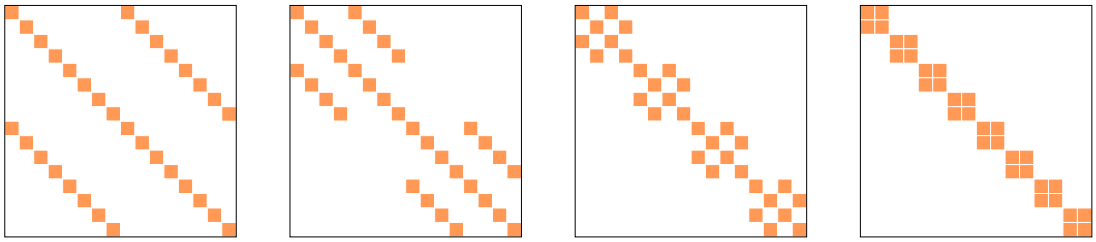
Each butterfly factor B_ℓ satisfies a fixed sparsity pattern, also called *support*. We define the support $S = \mathrm{supp}(A)$ of a matrix A as the binary matrix S with the same sparsity pattern as A but with 1 replacing all nonzero coefficients. The support S_ℓ of butterfly factors B_ℓ is defined as

$$S_\ell \coloneqq I_{2^{\ell-1}} \otimes \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} \otimes I_{n/2^\ell},$$

with I_k the identity matrix of order k and $\otimes$ the Kronecker product. This leads to highly sparse factors: each B_ℓ has exactly two nonzero elements on each row and on each column. The support of the butterfly factors for $n = 16$ are depicted in Figure 13.1.

The butterfly supports have the interesting property that for any subset of consecutive factors, the product between $X \coloneqq B_{\ell_0} \dots B_{\ell_1} \in \mathbb{R}^{n \times n}$ and $Y^T \coloneqq B_{\ell_1+1} \dots B_{\ell_2} \in \mathbb{R}^{n \times n}$, with $1 \leq \ell_0 \leq \ell_1 < \ell_2 \leq L$, can be written as

$$XY^T = \sum_{i=1}^{n} x_i y_i^T = \sum_{i=1}^{n} C_i, \tag{13.1}$$



(a) S_1 (b) S_2 (c) S_3 (d) S_4

Figure 13.1: Support of the butterfly factors, $n = 16$. Each factor S_i is an $n \times n$ matrix with binary entries. Zeros are represented in white, ones in orange.

where the n rank-one matrices $C_i := x_i y_i^T \in \mathbb{R}^{n \times n}$ (the rank-one components) associated with the columns of X and Y have disjoint supports [183, Lemma 2], that is,

$$\operatorname{supp}(C_i) \circ \operatorname{supp}(C_j) = 0_n, \quad i \neq j, \tag{13.2}$$

where $\circ$ denotes the Hadamard product and 0_n the $n \times n$ zero matrix. Moreover the matrices C_i partially retain the sparsity of the butterfly factors: a product of p consecutive factors has exactly 2^p nonzero elements on each row and on each column.

These crucial properties can be exploited to design an efficient factorization algorithm, which employs the SVD to compute the best rank-one approximations $x_i y_i^T$, with bounded complexity and error guarantees [183, 182, 260, 184]. We will see in Section 13.3 how to exploit these properties for the quantization of butterfly factors.

13.2 Applications of butterfly factorizations

Butterfly matrices are widely used in signal processing, machine learning, and numerical linear algebra. They appear, for instance, in the factorizations of the Hadamard and Fourier matrices [101].

Butterfly factorizations can be used to accelerate the solution of $Ax = b$ by Gaussian elimination by removing the need for pivoting. Originally proposed by Parker in 1995 [220], the idea is to preprocess A by premultiplying it with random butterfly transformations (RBT); Parker proved that the resulting matrix can be factored without pivoting with high probability. Note, however, that this does not guarantee a small growth factor ρ_n, and despite limited theoretical results [222], stability in finite precision remains an open issue. This RBT approach is therefore often combined with iterative refinement (see Chapter 10) to recover high accuracy [71, 189]. One limitation of butterfly factorizations is that they assume the matrix size $n = 2^L$ to be a power of two, which implies a significant overhead cost to handle matrices that do not satisfy this constraint. Lindquist et al. [191] propose a generalization of RBT that overcomes this limitation. While this RBT approach has encountered success for dense systems, it is much less suited for sparse ones. Baboulin et al. [72] investigate its application to sparse LU factorization and, while the method still allows for avoiding pivoting, it can increase the fill-in of the LU factors quite significantly in many cases.

Butterfly factorizations have also been used to design fast structured linear solvers. In the same spirit as block low-rank and hierarchical solvers (see Chapter 9), the idea is to approximate off-diagonal blocks of the matrix as butterfly factorizations. In particular, Liu et al. [192] have proposed a multifrontal solver that uses the HODBF format, the butterfly extension of the HODLR format, to compress the frontal matrices. Claus et al. [98] employ a composite approach

that uses HODBF only on the largest fronts. See [192] and references therein for more examples of applications using butterfly factorizations as a rank-structured compression format.

Finally, butterfly factorizations have also been used in machine learning. In a series of papers, Dao and co-authors have explored the use of butterfly factorizations to compress neural networks. In [105], they show that fully-connected layers can be replaced by matrices of the form $BPB'P'$ where B, B' denote butterfly factorizations and P, P' denote permutation matrices. In [106], they present the extension to more general "Kaleidoscope" matrices, which are of the form $(BB^*)^w$ for some power w. In [103], they propose "pixelated butterfly" matrices, a more hardware-friendly variant of butterfly factorizations that uses blocks of entries and approximates the layers as the sum of butterfly factors rather than their product, so that they can be applied in parallel; they add to this representation a low-rank term in order to improve its expressivity (see also the study [96] on sparse+low-rank representations). Finally, in [104], they propose "Monarch" matrices, which are obtained from butterfly factorizations by multiplying each half of their factors together; the resulting Monarch factorization is denser, requiring $O(n^{3/2})$ entries instead of $O(n \log n)$, but is more expressive and more hardware-friendly.

13.3 Quantization of butterfly factorizations

Given $L = \log_2(n)$ butterfly factors $B_1, \ldots, B_L \in \mathbb{R}^{n \times n}$ (that may have been obtained using an existing algorithm to approximate some target dense matrix), in **[42]**, we develop a method to quantize the factors $B_1, \ldots, B_L$ while minimizing the relative error

$$\frac{\|B_1 \ldots B_L - \widehat{B}_1 \ldots \widehat{B}_L\|}{\|B_1 \ldots B_L\|}. \tag{13.3}$$

The quantized factors $\widehat{B}_1, \ldots, \widehat{B}_L$ must have coefficients in $\mathbb{F}_t$, the set of floating-point numbers with t bits of significand, and retain the same support as the unquantized factors: $\operatorname{supp}(\widehat{B}_\ell) = \operatorname{supp}(B_\ell)$, $\ell = 1\colon L$.

We first discuss the case of a two-factor decomposition XY^T, in which case an optimal quantization $\widehat{X}\widehat{Y}^T$ can be found by solving a series of rank-one problems through our optimal rank-one quantization method presented in Algorithm 6.8. Indeed, exploiting the decomposition (13.1) into rank-one components *with disjoint supports*, the optimal quantization problem

$$\widehat{X}, \widehat{Y} \in \arg\min_{\widehat{X},\widehat{Y}} \|XY^T - \widehat{X}\widehat{Y}^T\|^2, \tag{13.4}$$

where $\widehat{X}, \widehat{Y}$ have coefficients in $\mathbb{F}_t$ and the same supports as X, Y, decouples into n independent optimal rank-one quantization problems. Its solution can thus be computed by n independent applications of Algorithm 6.8 and yields diagonal matrices $\Lambda, M \in \mathbb{R}^{n \times n}$ such that $\widehat{X} = \operatorname{round}(X\Lambda)$ and $\widehat{Y} = \operatorname{round}(YM)$.

We can then use this optimal two-factor quantization algorithm as a building block in a heuristic procedure to quantize a decomposition $B_1 \ldots B_L$ with more than two factors ($L > 2$). We can use different parenthesizations of the product to sequentially divide it in a series of two-factor products with disjoint rank-one components. Note that the quantizations of each individual two-factor product will be optimal, but there is no guarantee that the entire process is globally optimal.

We consider two heuristics based on different parenthesizations. The first heuristic uses a pairwise parenthesization $(B_1B_2)(B_3B_4)\ldots(B_{L-1}B_L)$ (assuming that L is even), and quantizations are performed on each pair of consecutive factors $(B_\ell B_{\ell+1})$ separately. If the number of factors L is odd, the last factor B_L is simply quantized as $\widehat{B}_L = \operatorname{round}(B_L)$. As this pairwise

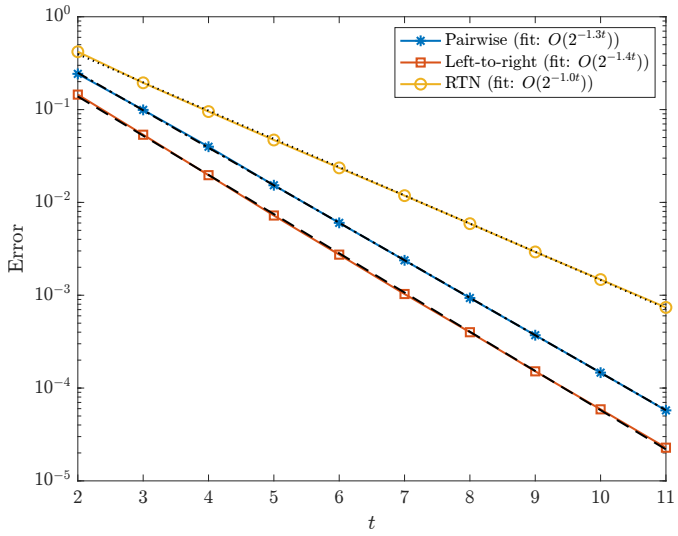


Figure 13.2: Relative quantization error (13.3) for butterfly factors with either the RTN baseline or pairwise or left-to-right algorithms, as a function of the number of bits t. The matrices are of order $n = 2^{16}$.

heuristic acts very locally, we also propose a second heuristic that can provide more accuracy. This second heuristic uses a parenthesization from left to right and quantizes the factors one by one in L steps. At step ℓ, the first $\ell - 1$ factors have already been quantized, and the remaining factors are parenthesized as $(MB_\ell)(B_{\ell+1} \dots B_L)$, where the current diagonal scaling M comes from the quantization at step $\ell - 1$. The quantized factor $\widehat{B}_\ell$ is obtained using the optimal two-factor quantization with $X = MB_\ell$ and $Y^T = B_{\ell+1} \dots B_L$.

The left-to-right heuristic is more expensive than the pairwise one. Indeed, the pairwise heuristic preserves the extreme sparsity of the butterfly matrices; in fact, it does not require forming any explicit intermediate matrix product and can directly work in-place by replacing the butterfly factors by their quantized version. In contrast, the left-to-right heuristic requires forming the explicit product of up to $L-1$ consecutive factors into Y^T, which is an almost dense matrix. However, this matrix can fortunately be formed only one row at a time, and so does not require much additional storage. In fact, both heuristics have the same space complexity in $O(2^t + n \log n)$. The time complexities are in $O(2^t n \log n)$ for the pairwise heuristic and in $O(2^t n^2)$ for the left-to-right one.

We illustrate the accuracy of these two heuristics in Figure 13.2, which plots the quantization error obtained for a number of bits t varying from 2 to 11, for a butterfly factorization of order $n = 2^{16}$ (this shows that the proposed heuristics, while limited to low precisions, can tackle large matrices). We compare three approaches: the naive baseline where each factor is separately quantized with round-to-nearest (RTN), and our two heuristic approaches based on the optimal two-factor quantization. We find that both heuristics significantly reduce the quantization error with respect to the RTN baseline and that, as expected, the more expensive left-to-right heuristic is more accurate than the pairwise one. Importantly, both heuristics achieve an accuracy that behaves significantly better than the $O(2^{-t})$ RTN accuracy: the pairwise and left-to-right heuristics approximately behave as $O(2^{-1.3t})$ and $O(2^{-1.4t})$, respectively. Thanks to this improved accuracy, our algorithms can achieve an accuracy equivalent to the RTN baseline with a lower precision; for example, the left-to-right heuristic can reduce the number of bits to $t/1.4$, which represents a $1 - 1/1.4 \approx 30\%$ reduction of storage for the same accuracy. In **[42]**, we also report the time cost of running these two heuristics, which we show to be very reasonable;

we are able to tackle large scale matrices (of order up to $2^{18} \approx 260000$) despite using a MATLAB code run on a single core laptop.

Given the success of butterfly factorizations to compress neural networks (see the previous Section 13.2), and the resilience of these networks to low precision quantization, a natural research perspective is to apply our quantization method to this context, and to extend it to other butterfly factorization variants tailored for neural networks; see Research Problem 16.27.

Chapter 14

Tensor approximations

Tensors are the higher-dimensional generalization of matrices. The size of a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ is exponential in its order d, so it is critical to develop compression methods that compute compressed representations, similar to low-rank approximations in the matrix case. However, unlike low-rank matrices, which can always be written under the form XY^T, there is not a unique format for low-rank tensors. Several formats have been proposed, some of the most popular ones being the tensor train [215], the Tucker [107], and the hierarchical Tucker [151, 141] formats.

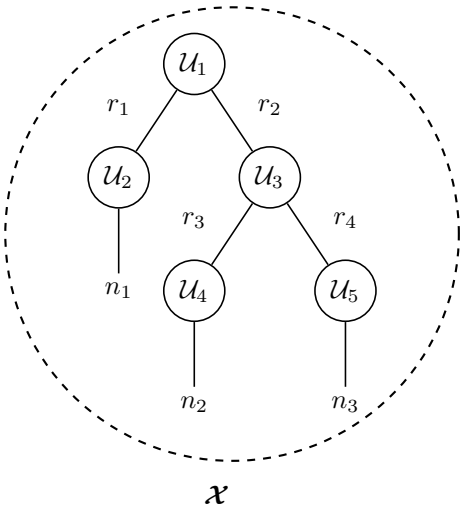


Figure 14.1: Tree tensor network representation

These different representations can be elegantly unified using tree tensor networks (TTN) [214] (see Figure 14.1). The nodes of a TTN $\boldsymbol{\mathcal{X}}$ represent tensors and the edges represent the dimensions of the tensor; thus the degree of a node is the order of the associated tensor. There are two types of edges: inner edges that connect two nodes, and outer edges that are only connected to one node. Two nodes connected by an inner edge represent the contraction of the corresponding tensors along the dimension that connects them. Therefore, the whole network represents a full tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ of order d, the number of outer edges of $\boldsymbol{\mathcal{X}}$.

Such compressed TTN representations can be computed via successive matrix LRA. For example, in Figure 14.1, node $\mathcal{U}_2$ can be computed by matricizing $\mathcal{X} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ along its

first dimension n_1 as $X \in \mathbb{R}^{n_1 \times n_2 \cdots n_d}$ and computing the matrix LRA $X \approx U_2 Y^T$, where $\mathcal{U}_2 \equiv U_2 \in \mathbb{R}^{n_1 \times r_1}$ and $Y \in \mathbb{R}^{n_2 \cdots n_d \times r_1}$. The remaining nodes can then be computed by reshaping Y back to tensor form and matricizing it along another dimension.

In this chapter, we do not aim to provide an exhaustive or detailed description of tensor formats and algorithms, for which we rather refer to existing surveys [174, 142]. We only provide a brief discussion of what the use of approximate LRA methods for computing TTN entails, in terms of numerical stability (Section 14.1) and potential use of approximation techniques such as mixed precision (Section 14.2).

The contributions described in this chapter are based on the following papers: **[39]** for Section 14.1.

14.1 Stability of TTN operations

While the stability of approximate tensor computations subject to low-rank truncation errors is well understood in exact arithmetic, it has been much less studied in finite precision arithmetic. This lack of analysis can be explained by mainly two reasons: first, the truncation errors tend to dominate the rounding errors when high precision is used, so that the latter have been traditionally neglected; and second, there is a wide range of different tensor formats, and their associated algorithms can be very complex to analyze. Importantly, the first reason is becoming less and less valid due to the growing prevalence of low precision arithmetics, which offer significant performance benefits on modern hardware. And the second reason makes it even more critical to develop a rounding error analysis precisely to identify which tensor algorithms are stable and which will become problematic in low precision.

We begin tackling these questions in **[39]**. In order for our analysis to be applicable to as many cases as possible, we propose a general framework based on TTNs and we define the essential operations that are needed to express some of the most common computations of interest. We consider two abstract kernels, Split, which replaces a node A by two connected nodes B and C, and Merge, which performs the converse operation. We assume these two kernels to be locally stable with respect to a precision parameter ε:

$$\widehat{B}\widehat{C} = \textsc{Split}(A) = A + E, \quad \|E\| \leq c\varepsilon\|A\|, \tag{14.1}$$

$$\widehat{\mathcal{A}} = \textsc{Merge}(\mathcal{B}, \mathcal{C}) = \mathcal{B}\mathcal{C} + \mathcal{E}, \quad \|\mathcal{E}\| \leq c\varepsilon\|\mathcal{B}\|\|\mathcal{C}\|. \tag{14.2}$$

We then carry out an error analysis of an abstract computation formed of a sequence of such Split and Merge operations. Our analysis reveals a key condition for the computation to be stable: the norm of the tensor should be tightly concentrated around a single node of the network, a property that we formalize as follows.

Definition 14.1. *Consider a network $\boldsymbol{\mathcal{X}}$ composed of n nodes $\mathcal{U}_1, \ldots, \mathcal{U}_n$. $\boldsymbol{\mathcal{X}}$ is said to be α-normalized with respect to a node $\mathcal{U}_j$ if there exists a vector of constants $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$ such that $\|\mathcal{U}_j\| \leq \alpha_j \|\boldsymbol{\mathcal{X}}\|$ and such that the matricization U_i of all the other nodes $\mathcal{U}_i$, $i \neq j$, in the direction of $\mathcal{U}_j$, satisfies, for any matrix M of compatible dimensions, $\|U_i M\| \leq \alpha_i \|M\|$.*

In this definition, the matricization of a node $\mathcal{U}_i$ in the direction of $\mathcal{U}_j$ refers to the matricization of the tensor $\mathcal{U}_i$ along its dimension that belongs to the path connecting it to $\mathcal{U}_j$: for example in Figure 14.1, the matricization of $\mathcal{U}_3$ in the direction of $\mathcal{U}_1$ is a matrix $U_3 \in \mathbb{R}^{r_2 \times r_3 r_4}$.

We are now ready to state the following result, which identifies conditions for an abstract TTN computation to be stable.

Theorem 14.1 (Theorem 3.2 in **[39]**). *Let $\widehat{\boldsymbol{\mathcal{X}}}$ be obtained by applying a sequence of p operations* Split *or* Merge *to $\boldsymbol{\mathcal{X}}$. Under the assumptions (14.1) and (14.2), and assuming that at each step k of the computation, the intermediate network $\boldsymbol{\mathcal{X}}_k$ is $\alpha^{(k)}$-normalized as defined in Definition 14.1 with $\alpha^{(k)} \in \mathbb{R}^{n_k}$, we have*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq s\varepsilon\|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2) \tag{14.3}$$

with

$$s = \sum_{k=0}^{p-1} cr_k \prod_{i=1}^{n_k} \alpha_i^{(k)},$$

where r_k is the largest inner dimension of $\boldsymbol{\mathcal{X}}_k$ and c is the constant in (14.1) and (14.2).

The bound (14.3) can be simplified to

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \lesssim pr\alpha^n c\varepsilon\|\boldsymbol{\mathcal{X}}\|,$$

with $r = \max_k r_k$, $\alpha = \max_{i,k} \alpha_i^{(k)}$, and where

$$n = \max_k \#\left\{i = 1\colon n_k,\ \alpha_i^{(k)} \neq 1\right\}$$

is the maximum number of nodes that are not 1-normalized at any step k. Therefore, Theorem 14.1 shows that the error grows linearly with the kernel error $c\varepsilon$, the largest inner dimension r of the network, the number of operations p, and the term α^n, which is exponential in the number of nodes n that are not 1-normalized. Therefore, if the kernels are stable (small c), the main condition for the overall network computation to also be stable is that α^n is small.

There are two important examples of TTN when this condition is satisfied. The first is when all nodes of the network except one are orthonormal; in this case the network is α-normalized with $\alpha = e$, the vector of all ones, and thus $\alpha^n = 1$. This first example shows that if the orthonormality of all nodes except one is maintained throughout the computation, numerical stability is guaranteed.

The second example is a TTN $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{Y}} + \boldsymbol{\mathcal{Z}}$ that is the sum of two e-normalized TTNs $\boldsymbol{\mathcal{Y}}$ and $\boldsymbol{\mathcal{Z}}$; this can arise in computations involving TTN arithmetic. In this case, the inner nodes of $\boldsymbol{\mathcal{X}}$ become the block diagonal concatenation of the corresponding nodes in $\boldsymbol{\mathcal{Y}}$ and $\boldsymbol{\mathcal{Z}}$, and thus remain orthonormal. On the other hand, the outer nodes (that is, the leaves) of $\boldsymbol{\mathcal{X}}$ become the horizontal concatenation of the corresponding nodes in $\boldsymbol{\mathcal{Y}}$ and $\boldsymbol{\mathcal{Z}}$, and therefore lose their orthonormality; they are, however, $\sqrt{2}$-normalized. In this case, we thus have $\alpha^n = 2^{\ell/2}$, where ℓ is the number of leaves. For some TTN formats, ℓ is small (for example, $\ell = 2$ in the tensor train format), and so stability is once more guaranteed. For other formats (such as Tucker), $\ell = O(d)$ and so stability is only guaranteed for low order tensors. In practice, however, the constant $2^{\ell/2}$ is pessimistic because it does not take into account the specific structure of TTN given by $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{Y}} + \boldsymbol{\mathcal{Z}}$; see Research Problem 16.21.

In any case, Theorem 14.1 applies to a wide range of TTN computations that can be expressed in our framework, and shows that their stability is conditioned on orthonormality being maintained throughout the computation. In view of this conclusion, a second contribution of **[39]** is to propose a stable TTN rounding algorithm that is careful to maintain orthonormality. Essentially, the algorithm consists in truncating the inner dimensions one by one, while transferring the non-orthonormality throughout the network from node to node; see Algorithm 5.1 in **[39]** for a detailed description.

The algorithm works for any tree topology of tensor and can thus be applied to a wide range of tensor formats. When applied to specific tree topologies, it can be shown to be equivalent

to existing algorithms such as TTSVD [215] or HODSVD [243], which we have thus proven to be stable in finite precision arithmetic. We compare this stable rounding algorithm with the Gram LRA–based rounding, a popular choice for hierarchical Tucker [141] and tensor train [58] topologies, despite being unstable: as discussed in Section 6.6 and **[9]**, the accuracy of Gram LRA is proportional to $\sqrt{u}$, the square root of the machine precision u. We confirm experimentally that our stable algorithm can significantly improve the accuracy of the rounding in finite precision arithmetic, and can reliably exploit a precision as low as $u \approx \varepsilon$, the truncation threshold and target accuracy, whereas Gram LRA–based rounding must set $u \approx \varepsilon^2$. Therefore our algorithm is better suited to take advantage of all the performance benefits of lower precision arithmetics on modern hardware. While the asymptotic complexity of both approaches is comparable, Gram LRA is however more parallel: this is because our algorithm must transfer the non-orthonormality throughout the network and cannot, for example, process independent branches of the network in parallel. This observation therefore motivates a performance study to compare the practical performance of both algorithms on various computer architectures depending on their parallel and arithmetic precision environment; see Research Problem 16.22.

14.2 Approximate tensor computations

As mentioned, tensor computations heavily rely on matrix operations and in particular matrix LRA. Therefore, many of the approximate LRA methods discussed in Chapter 6 extend to tensors. To give a few examples:

- We have already mentioned Gram LRA (Section 6.6) in the previous section. It has indeed been shown to be particularly useful for the rounding of tensors. It was initially proposed for the compression of the tensor train (TT) format [216], and even though a stable rounding approach was later proposed as an alternative [215], Gram LRA remains highly efficient; the recent work of Al Daas et al. [58] presents a high performance parallel implementation where Gram LRA is a key component. Gram LRA is also a central tool for the rounding of hierarchical Tucker tensors [141, 175].

- There is a large body of literature on using randomized algorithms for computing tensor decompositions. In particular, the randomized range finder method described in Section 6.2 can be applied to compute the matrix LRA underlying tensor decompositions: see Ahmadi-Asl et al. [56] and references therein; in particular, Minster et al. [199] describe a randomized HOSVD method based on the fixed-accuracy range finder Algorithm 6.4. Randomized algorithms have also been proposed for the tensor rounding operation in Al Daas et al. [57].

- As explained in Section 6.4, randomized LRA methods are especially amenable to the use of multiword arithmetic (Chapter 5). The same remark applies in the tensor case: multiword arithmetic has been leveraged for accelerating randomized HOSVD by Ootomo and Yokota [212].

- In Section 6.5 we have described an iterative refinement method for LRA that we proposed in **[22]**. This method is actually quite general and can apply to both matrices and tensors. In particular, the refinement leads to tensor arithmetic operations and thus to the growth of the inner dimensions, which is mitigated by the recompression step (line 5 in Algorithm 6.6), which corresponds to the tensor rounding operation. We perform experiments on various tensor formats that confirm that high accuracy can be achieved even though low precision operations are used for the tensor compression.

Some other recent advances on accelerating or improving matrix LRA methods have not, to our knowledge, yet been extended to tensors. In particular, it seems most promising to investigate adaptive precision (see Research Problem 16.23) and optimally quantized (see Research Problem 16.24) tensor decompositions.

Chapter 15

Neural networks

Neural networks are a fundamental tool appearing in a wide range of artificial intelligence applications. Approximate linear algebra computations are of high interest for neural networks, because neural networks are both computationally intensive yet also quite resilient to errors. Despite the widespread adoption of approximate computing techniques in neural networks, there remains a significant gap in our mathematical understanding of the impact that these approximations have on the behavior of neural networks.

In this chapter, we aim to start reducing this gap by developing error analyses that investigate the propagation of errors in feedforward neural networks. Specifically, we model the computation of interest as

$$y_\ell = \phi_\ell(A_\ell y_{\ell-1}), \qquad \ell = 1\colon L, \tag{15.1}$$

where $y_0 \in \mathbb{R}^{n_0}$ is the input vector, L is the number of layers, $A_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ are L weight matrices, and $\phi_\ell : \mathbb{R}^{n_\ell} \mapsto \mathbb{R}^{n_\ell}$ are L nonlinear activation functions. Typical activation functions include ReLU and tanh.

We mainly focus on multilayer perceptron (MLP) networks, whose fully-connected layers lead to dense weight matrices A_ℓ. However, our analyses can also apply to convolutional networks (CNN), by interpreting the convolutional layer operators as sparse weight matrices; a dedicated analysis for CNNs might however lead to sharper bounds or specific insights. Finally, due to their efficiency to train large language models (LLMs), transformer networks have become widely popular. Our analyses do not directly apply to transformers but could be extended; see Research Problem 16.29.

We investigate both backward and forward errors; a componentwise analysis of the latter allows us to derive a new mixed precision algorithm for neural network inference.

The contributions described in this chapter are based on the following papers: **[44]** for Section 15.1; **[43]** for Section 15.2.

Componentwise notations This chapter will make intensive use of Hadamard's componentwise notations. We denote as $x \circ y$ and $x \oslash y$ the vectors resulting from the componentwise multiplication and division of x and y, respectively. The Hadamard product of a matrix with a vector (denoted $x \circ A$ or $A \circ x$) multiplies the rows of the matrix by the components of the vector. We also generalize the γ notation to vectors: γ_x denotes the vector whose components are γ_{x_i}. Inequalities between vectors $x \leq y$ or matrices $A \leq B$ of identical dimensions hold componentwise; moreover, an inequality $A \leq x$ between a matrix $A \in \mathbb{R}^{m\times n}$ and a vector $x \in \mathbb{R}^m$ applies to each row of A componentwise, that is, $a_{ij} \leq x_i$ for all i, j. Finally, operations (such as $x + c$) and inequalities (such as $x < c$) with scalar constants $c \in \mathbb{R}$ also apply componentwise.

15.1 Backward error analysis

In [44], we develop a *backward* error analysis of the computation (15.1). Our goal is thus to recast the errors incurred in such a computation as perturbations on the network's parameters, specifically the weight matrices.

Assuming standard floating-point arithmetic, the rounding errors incurred in the matrix–vector products $v_\ell = A_\ell y_{\ell-1}$ directly satisfy the desired form: the computed $\widehat{v}_\ell$ satisfies indeed

$$\widehat{v}_\ell = (A_\ell + \Delta A_\ell)\widehat{y}_{\ell-1}, \quad |\Delta A_\ell| \leq \gamma_{n_{\ell-1}}|A_\ell|. \tag{15.2}$$

For the errors incurred in the evaluation of the activation function $y_\ell = \phi_\ell(v_\ell)$, we use a similar model to the standard Model 2.1 for elementary floating-point operations: we assume that the computed $\widehat{y}_\ell$ satisfies

$$\widehat{y}_\ell = \phi_\ell(\widehat{v}_\ell) + \Delta y_\ell, \quad |\Delta y_\ell| \leq \gamma_c|\phi_\ell(\widehat{v}_\ell)| \tag{15.3}$$

for some constant c. Available implementations of typical activation functions usually satisfy this model for small values of c. The perturbation Δy_ℓ applies to the output of ϕ_ℓ; in order to recast it to the input, we introduce the relative condition number of a function f:

$$\kappa_f(x) = |xf'(x)| \oslash |f(x)|. \tag{15.4}$$

Assuming that $\kappa_{\phi_\ell}(\widehat{v}_\ell) > 0$, we can then rewrite (15.3) as

$$\widehat{y}_\ell = \phi_\ell(\widehat{v}_\ell + \Delta v_\ell), \quad |\Delta v_\ell| \leq \gamma_c|\phi_\ell(\widehat{v}_\ell)| \oslash \kappa_{\phi_\ell(\widehat{v}_\ell)}. \tag{15.5}$$

We can now state the following backward error result.

Theorem 15.1 (Theorem 2 in [44])**.** *Consider a neural network with L layers whose output is given by (15.1). Assuming (15.2) and (15.5), the computed $\widehat{y}_L$ satisfies*

$$\widehat{y}_L = \phi_L((A_L + \Delta A_L)\phi_{L-1}((A_{L-1} + \Delta A_{L-1})\dots(A_2 + \Delta A_2)\phi_1((A_1 + \Delta A_1)y_0)\dots)) \tag{15.6}$$

where, for $\ell = 1\colon L$,

$$|\Delta A_\ell| \leq \gamma_{n_{\ell-1}+c/\kappa_{\phi_\ell}(A_\ell\widehat{y}_{\ell-1})} \circ |A_\ell|. \tag{15.7}$$

Theorem 15.1 shows that the computed output $\widehat{y}_L$ can be expressed as the exact output of a network with perturbed weights. We can conclude that the backward stability of such a neural network computation is conditioned on the term $\gamma_{n_{\ell-1}+c/\kappa_{\phi_\ell}(A_\ell\widehat{y}_{\ell-1})}$ being close to the unit roundoff for all layers. In the following, we abbreviate this term simply as $\gamma_{n+c/\kappa}$ for the ease of discussion.

The first term that may pose problems to stability is n, the size of the layers. Indeed, since n can be large for complex models such as LLMs, the resulting rounding error accumulation in the matrix–vector products could be problematic, especially given the low precisions typically used. Fortunately, multiple factors come to mitigate this risk. First, as we have discussed in Chapter 3, the statistical distribution of rounding errors leads to reduced probabilistic error bounds. We prove indeed in [44] (Theorem 3) that, under Model 3.1, n can be replaced by $\sqrt{n}$. Since we have seen in Section 3.2 that stochastic rounding enforces this model to hold, this provides a theoretical explanation of the success of this rounding mode in machine learning [147, 125]. Moreover, we also prove in [44] (Theorem 5) that $\sqrt{n}$ can be further reduced to a constant independent of n if the weight matrices have mean zero coefficients, that is, if they satisfy Model 3.2 with $\mu_x = 0$. We illustrate experimentally that this property is indeed approximately satisfied for networks trained on some typical datasets. Yet another mitigation factor is the fact that AI-specialized

hardware, such as GPU tensor cores, often provide high precision accumulators; as analyzed in Section 4.1, in this case $\gamma_n \approx nu$ can be replaced by $2u_{\text{low}} + nu_{\text{high}}$. Finally, even if all of the above does not apply, there always remains the possibility of using in the underlying inner products a summation algorithm that reduces error accumulation. Blocked summation has for example been used by Wang et al. [248].

This leaves the second term, c/κ. Importantly, for nonlinear functions, the condition number κ can be smaller than 1, and can in fact be arbitrarily close to 0 or even exactly 0. For example, $\kappa = 0$ when the activation function is constant, which is notably the case for the negative half of ReLU. Therefore, the term c/κ can be arbitrarily large and a small backward error cannot be guaranteed for any input. We however note that this issue is largely an artifact of forcing the error to be expressed as a backward error. In fact, we will prove in the next Section 15.2 that the computation is *forward* stable, in the sense that it satisfies a forward error bound of the same order as if it was backward stable [162, sect. 1.6] (that is, of order the condition number times the unit roundoff). Informally, this can be seen by multiplying the backward error bound by κ, yielding $\gamma_{n\kappa+c}$; this however neglects to take into account the condition number of the matrix–vector product.

15.2 Forward error analysis and a mixed precision accumulation algorithm

In **[43]**, we carry out a forward error analysis of (15.1). The analysis is similar in spirit to the backward error one of the previous section, but presents two significant differences. First, as anticipated above, we will prove forward stability whereas backward stability is not guaranteed. Second, we use a more general error model that will suggest a mixed precision strategy.

Model 15.1. *We assume that $\widehat{y}_0 = y_0$ and that each computed $\widehat{y}_\ell$ satisfies*

$$\widehat{y}_\ell = \phi_\ell\big((A_\ell \circ (1 + \Delta A_\ell))\widehat{y}_{\ell-1}\big) \circ (1 + \Delta\phi_\ell), \quad |\Delta A_\ell| \le \varepsilon_\ell^A, \quad |\Delta\phi_\ell| \le \varepsilon_\ell^\phi, \tag{15.8}$$

where $\Delta A_\ell \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, $\Delta\phi_\ell \in \mathbb{R}^{n_\ell}$, $\varepsilon_\ell^A \in \mathbb{R}^{n_\ell}$ is a nonnegative vector whose components bound the backward errors incurred in the evaluation of the inner products with the rows of A_ℓ, that is, $|(\Delta A_\ell)_{ij}| \le (\varepsilon_\ell^A)_i$, and $\varepsilon_\ell^\phi \in \mathbb{R}^{n_\ell}$ is a nonnegative vector whose components bound the forward errors incurred in the evaluation of ϕ_ℓ.

Through its precision parameters ε_ℓ^A and ε_ℓ^ϕ, Model 15.1 is indeed very general. First, we can use different precisions for the inner products (ε^A) and the activation functions (ε^ϕ). Second, we can vary these precisions between layers (different ℓ). Lastly, we can also vary these precisions across different components (different $(\varepsilon_\ell)_i$). As our analysis will reveal, this flexibility in choosing the precisions allows for devising a meaningful mixed precision strategy.

Theorem 15.2 (Theorem 2.4 and Corollary 2.5 in **[43]**). *Let $y_\ell = \phi_\ell(A_\ell y_{\ell-1})$ be computed inexactly such that the computed $\widehat{y}_\ell$ satisfies Model 15.1. Then, we have*

$$\widehat{y}_\ell = y_\ell \circ (1 + \Delta y_\ell), \quad |\Delta y_\ell| \le \varepsilon_\ell^y,$$

where ε_ℓ^y satisfies the recurrence relation

$$\varepsilon_\ell^y = \kappa_{\phi_\ell}(v_\ell) \circ \kappa_{v_\ell} \circ \big(\varepsilon_\ell^A + \|\varepsilon_{\ell-1}^y\|_\infty(1 + \varepsilon_\ell^A)\big) \circ (1 + \varepsilon_\ell^\phi) + \varepsilon_\ell^\phi,$$

where κ_{ϕ_ℓ} is the condition number of ϕ_ℓ (see (15.4)) and κ_{v_ℓ} is the condition number of the matrix–vector product $v_\ell = A_\ell y_{\ell-1}$ defined as $\kappa_{v_\ell} = |A_\ell||y_{\ell-1}| \oslash |A_\ell y_{\ell-1}|$. To first order, the

computed final output of the network $\widehat{y}_L$ therefore satisfies

$$\widehat{y}_L = y_L \circ (1 + \Delta y_L), \quad |\Delta y_L| \lesssim \varepsilon_L^y,$$

with

$$\|\varepsilon_L^y\|_\infty = \sum_{\ell=1}^{L} \left[\Big(\prod_{k=\ell+1}^{L} \|\kappa_{\phi_k}(v_k) \circ \kappa_{v_k}\|_\infty \Big) \Big(\|\kappa_{\phi_\ell}(v_\ell) \circ \kappa_{v_\ell} \circ \varepsilon_\ell^A\|_\infty + \|\varepsilon_\ell^\phi\|_\infty \Big) \right]. \tag{15.9}$$

One lesson that can be drawn from bound (15.9) is that the errors ε_ℓ^ϕ from the activation functions appear in the infinity norm. This suggests that it is meaningless to vary the precision of the activations between different components, because only the maximum error component from the previous layer is propagated; thus we may as well compute all the components in the same precision. On the other hand, the term $\|\kappa_{\phi_\ell}(v_\ell) \circ \kappa_{v_\ell} \circ \varepsilon_\ell^A\|_\infty$ tells quite a different story. The precisions ε_ℓ^A are multiplied componentwise by the condition number

$$\kappa_\ell := \kappa_{\phi_\ell}(v_\ell) \circ \kappa_{v_\ell}, \tag{15.10}$$

and so we should try to balance each component of $\kappa_\ell \circ \varepsilon_\ell^A$ to minimize their maximum. This therefore suggests that we should choose the precision of the inner product with the ith row of A_ℓ to be inversely proportional to the ith component of κ_ℓ. This represents a good opportunity to introduce mixed precision in the computation: we expect the components of κ_ℓ to exhibit wide variations in magnitude. Indeed, as mentioned before, for typical activation functions such as ReLU or tanh, κ_{ϕ_ℓ} can be arbitrarily small or even equal to zero. The corresponding inner products can be computed in very low precision, while still maintaining a high accuracy on the overall computation.

In order to leverage this observation into a practical algorithm, one critical difficulty must be overcome: we do not know the values of κ_ℓ! Indeed, recall that $\kappa_\ell = \kappa_{\phi_\ell}(v_\ell) \circ \kappa_{v_\ell}$ depends on $v_\ell = A_\ell y_{\ell-1}$; therefore, computing κ_ℓ and thus v_ℓ in high precision would defeat the purpose of using mixed precision, since v_ℓ is precisely the result of the matrix–vector product that we aim to accelerate. Moreover, for any layer ℓ, v_ℓ depends in particular on y_0, the input of the network, so the precision choices depend on the input and cannot be reused across different inputs.

Fortunately, we do not need a very accurate computation of κ_ℓ: estimating its order of magnitude is sufficient to decide which precision to use. This key observation is at the foundation of a practical method outlined in Algorithm 15.1. For each layer, we first compute v_ℓ in precision u_{low}, that is, we perform the entire matrix–vector product in low precision. We also compute the corresponding activation $y_\ell = \phi_\ell(v_\ell)$ in low precision. Then, we use these approximate v_ℓ and y_ℓ to estimate κ_ℓ; while κ_{ϕ_ℓ} can be readily computed, computing $\kappa_{v_\ell} = (|A_\ell||y_{\ell-1}|) \oslash |v_\ell|$ would be too costly since it would require an extra matrix–vector product for evaluating the numerator. We can however avoid this computation, thanks to the key observation that the variations in magnitude of κ_{v_ℓ} are mostly due to variations of the denominator (we support this argument experimentally in **[43]**). Finally, for each component $(\kappa_\ell)_i$, we check whether it was safe to compute $(v_\ell)_i$ in low precision. Given some tolerance parameter τ, if $(\kappa_\ell)_i \leq \tau$, the component $(v_\ell)_i$ computed in low precision is kept, whereas if $(\kappa_\ell)_i > \tau$, $(v_\ell)_i$ is recomputed in high precision u_{high}.

This approach will therefore work best in situations where most components can be computed in low precision, and high precision is only needed to recompute a few of the most sensitive components. Indeed, if the criterion leads to too many components needing to be recomputed, this mixed precision approach may end up being more expensive than simply computing everything in high precision from the start. In order to quantify this more precisely, let us thus focus on

Algorithm 15.1 Neural network inference with mixed precision accumulation

Input: $A_1, \dots, A_L$, the weight matrices; y_0, the input vector; τ, a tolerance controlling the precision choice; $u_{\mathrm{low}}, u_{\mathrm{high}}$, the precisions.

Output: y_L, the output of the network.

for $\ell = 1, \dots, L$ **do**
  Compute $v_\ell = A_\ell y_{\ell-1}$ in precision u_{low}.
  Compute $y_\ell = \phi_\ell(v_\ell)$ in precision u_{low}.
  Compute $\kappa_{\phi_\ell}(v_\ell) = |v_\ell \circ \phi_\ell'(v_\ell)| \oslash |y_\ell|$ in precision u_{low}.
  Compute $\kappa_\ell = \kappa_{\phi_\ell} \oslash |v_\ell|$ in precision u_{low}.
  for every component $(\kappa_\ell)_i$ **do**
    if $(\kappa_\ell)_i > \tau$ **then**
      Recompute $(v_\ell)_i = (A_\ell y_{\ell-1})_i$ in precision u_{high}.
      Recompute $(y_\ell)_i = \phi_\ell((v_\ell)_i)$ in precision u_{high}.
      Requantize $(y_\ell)_i$ back to precision u_{low}.
    end if
  end for
end for

the cost of the matrix–vector products, which are likely to be the computational bottleneck. Let c_{low} be the cost of performing all the matrix–vector products (across all layers) in uniform precision u_{low}, and let c_{high} be the corresponding cost when using uniform precision u_{high} instead. Let $\rho \in [0, 1]$ be the fraction of components—and thus of inner products—that need to be recomputed in precision u_{high}. Then the cost of the mixed precision Algorithm 15.1 is

$$c_{\mathrm{mixed}} = c_{\mathrm{low}} + \rho c_{\mathrm{high}} = \left(\frac{c_{\mathrm{low}}}{c_{\mathrm{high}}} + \rho \right) c_{\mathrm{high}}, \tag{15.11}$$

Thus, for c_{mixed} to be less than c_{high}, we must have the condition $c_{\mathrm{low}}/c_{\mathrm{high}} + \rho < 1$. In other words, the mixed precision cost will be less than the high precision one if the costs ratio between the low and high precision is sufficiently small, and the fraction of components that need to be recomputed in high precision is also sufficiently small.

Figure 15.1 illustrates the potential of Algorithm 15.1 for accelerating the inference using a three-layer MLP network with ReLU activations, trained on the fashion MNIST dataset (see **[43]** for more experiments with different numbers of layers, tanh activations, and other datasets). We use fp16 as u_{high} and fp8 (e4m3) as u_{low}, and we compare the test accuracy of Algorithm 15.1 for various choices of tolerance τ with the fp8 and fp16 uniform precision baselines. The figure shows that, as τ decreases, the fraction ρ of inner products recomputed in fp16 increases, as does the test accuracy. Strikingly, with $\tau = 0.1$, our mixed precision algorithm reaches the same (actually slighty better) test accuracy than the uniform fp16 baseline, while only (re)computing 20% of the inner products in fp16; the other 80% are computed in fp8! Assuming $c_{\mathrm{low}}/c_{\mathrm{high}} = 0.5$, the cost formula (15.11) yields $c_{\mathrm{mixed}} = 0.7 c_{\mathrm{high}}$, that is, we can expect a 30% time reduction.

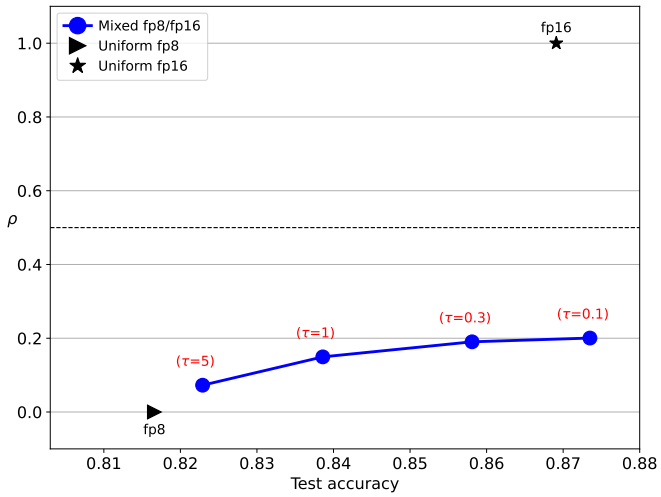


Figure 15.1: Test accuracy (x-axis) depending on the fraction ρ of inner products (re)computed in fp16. We compare our mixed precision Algorithm 15.1 with varying τ with the uniform fp8 ($\rho = 0$) and fp16 ($\rho = 1$) baselines.

Chapter 16

Conclusion

I conclude this manuscript by presenting some research directions that aim to fulfill the initial objectives discussed in Chapter 1: making approximate computing the standard, default mode of computing at exascale. Indeed, while it is now clear that approximations will have an important role to play in the future of computing, there still remains a long road before they can be integrated seamlessly, safely, and efficiently in mainstream general purpose computing.

In Section 16.1, I first discuss in general terms some of the most pressing challenges and put forward some of the methods that I expect to become more and more important. Then, in Section 16.2, I list several concrete research problems that I think are worth investigating in the near future; most of them build upon the contributions previously presented in this manuscript.

16.1 Future challenges and research directions

Exascale methods for exascale computers While exascale computers are starting to appear, we do not yet have "exascale" methods that can exploit them efficiently. Indeed, despite the size and power of exascale computers, we cannot afford to solve today's problems exactly due to their extreme size and the high computational cost of exact computations. We must therefore resort to approximate computing to reduce the size of the problems and the complexity of the computations. However, while approximate computing methods do significantly outperform exact methods, they are also much less efficient at exploiting exascale computers. This is due to a combination of several factors. Approximations tend to reduce flops by a much larger factor than storage, and so approximate methods are much more memory- or communication-bound than exact ones. Approximations lead to unpredictable and unbalanced workloads, that are much harder to balance and schedule in parallel environments. These issues are even more critical on accelerators, which require high amounts of concurrency and large regular workloads. Therefore, to harness both the power of exascale computers and that of approximate computing, it is necessary to develop new approximate methods that are specifically designed for being efficient on massively parallel computers. As such, communication-avoiding, randomized, and other methods improving the efficiency and scalability of the computations are likely to become even more prevalent.

Specialized hardware Since the IEEE 754 standard for floating-point arithmetic was established in 1985, the relative stability of the floating-point landscape has allowed numerical algorithms to be developed almost independently of the target arithmetic and even hardware. However, the situation is changing with the emergence of specialized mixed precision hardware, primarily designed for artificial intelligence applications. Such specialized hardware offers truly

unprecedented performance, but only for a few dedicated tasks (such as matrix multiplication) and only when using low precision (16-bit and lower). Moreover, the rise of such specialized hardware is also concomitant with the decreasing support of higher precision arithmetics; general purpose scientific computing is threatened by a gradual disappearance of 64-bit arithmetic in modern hardware. As such, multiword arithmetic is likely to become a crucial tool for emulating higher precisions while preserving efficiency. Moreover, this not merely a question of performance: it is necessary to take into account the specific numerical features of such specialized hardware when designing algorithms, especially approximate ones, in order to maximize accuracy and ensure robustness. For example, one should take advantage of high precision accumulators whenever possible, and one should employ scaling in order to maximize the utilization of the range of representable values.

Automatization and industrialization The industrialization of approximate methods, that is, their use in a realistic setting by an end-user with little knowledge on approximate computing and with no intervention from the developer/designer of the method, is extremely challenging. First, approximate methods tend to introduce additional parameters that affect their efficiency and/or accuracy, and that need to be controlled, or even carefully tuned, by the user. Second, some approximate methods, by design, only work on a subset of problems and fail on the rest (for example, if the problem is too ill conditioned), so that they should not be used blindly. Third, the behavior (both in terms of performance and accuracy) of approximate methods is hard to predict and may depart significantly from theory. All of these factors contribute to making it very challenging for end-users to be able to safely and easily rely on approximations. To tackle these challenges, it is therefore necessary to develop automated strategies, that incorporate both a priori predictors, to help decide which method should be used and with which parameters, and a posteriori checks, to verify that the method succeeded or on the contrary detect failures and adopt suitable fallback strategies. A priori tools are especially challenging to develop since the numerical behavior of many methods is still not well understood. This motivates the development of modular error analyses, to help refine our understanding of the role of each parameter, and of probabilistic error analyses, to ensure the sharpness of these a priori estimates. These analytical tools should be coupled with practical ones, such as condition number estimators.

Modular approximations As mentioned above, the behavior of many complex numerical algorithms is still not well understood. While some approximate methods were able to be developed by combining educated intuitions and empirical heuristics, this methodology will be increasingly unlikely to yield meaningful results, as the number of available precisions and types of approximations grows. To face this challenge, I advocate for a systematic approach that makes algorithms and their analysis modular by breaking them into independent smaller parts (the "modules"). The granularity of the parts is context-dependent: in some cases, it may suffice for each part to be standard linear algebra kernels whose behavior we understand, while in other cases, descending to the level of scalar instructions may be beneficial. This modular approach presents several advantages for approximate computing: from a theoretical point of view, analyzing modular algorithms where each module is parameterized by an independent precision parameter allows for identifying the role of this specific module on the behavior of the algorithm; this methodology can be used to discover new mixed precision strategies. Beyond that, modular computing makes it natural to combine different approximate techniques, such as mixed precision, low-rank approximations, randomization, or multiword arithmetic. This leads to questions (and hopefully answers) about the stability and efficient implementation of such combinations.

Adaptive approximations The behavior and potential of approximations strongly depend on the input data that they are applied to. In order to make the most out of approximations regardless of the input, adaptive methods should be used. Adaptive methods seek to dynamically detect numerical structure in the data at hand and to take advantage of it whenever possible. This notably includes adaptive precision algorithms, to which we have dedicated Chapter 11 of this manuscript. These algorithms have already been proven successful in a few specific contexts, but they have the potential to be effective in a much wider range of computations and applications, thanks to their many strengths. They can exploit a continuum of precisions and are thus well suited to harness emerging hardware where arithmetics range from 64-bit to just 4-bit. They can deliver a continuum of accuracies and thus offer flexible and versatile performance–accuracy tradeoffs. Perhaps most importantly, they can be used in a fully automated way: it suffices to give them a target accuracy and they will do the rest, that is, exploit as much approximations as possible while provably meeting the target. Moreover, when applied to adversarial inputs that do not present any approximation opportunities, their behavior often simply becomes equivalent to the standard non-approximate (uniform precision) approach. This makes them robust not only in terms of accuracy but also performance. For these reasons, I believe adaptive algorithms to be an ideal tool for black-box, industrial approximate computing.

16.2 Some research problems

16.2.1 Probabilistic error analysis

Research Problem 16.1 (Sharper probabilistic bounds for matrix factorizations)**.** While the constant $\widetilde{\gamma}(\lambda)$ appearing in probabilistic error bounds (see Theorem 3.1) is sharp for summation, inner products, and matrix–vector products (see Figure 3.1), there are cases where it still remains pessimistic. In particular, the backward error for matrix factorizations is in practice independent of the matrix size. We have developed in Section 3.3 an analysis that takes into account the data distribution and produces sharper bounds for zero-mean data. Unfortunately, these bounds are not applicable to matrix factorizations, since the entries of the LU factors of a matrix with random independent entries are no longer independent. Besides, the absence of error growth is observed for matrices with entries in both $[0, 1]$ and $[-1, 1]$, so a different mechanism seems to be at play. Providing an explanation for this phenomenon is an important open problem.

Research Problem 16.2 (Probabilistic analysis of fast matrix multiplication)**.** Fast matrix multiplication, like Strassen's algorithm [239], can compute $n \times n$ matrix products in $O(n^\omega)$ complexity with $\omega < 3$. Unfortunately, in floating-point arithmetic, they lead to a larger error growth; for example, the worst-case error bound for Strassen's algorithm grows as $n^{\log_2 12} \approx n^{3.6}$ [162, Thm. 23.2]. However, in practice, this bound is quite pessimistic; a probabilistic analysis of such algorithms might lead to sharper error bounds. Note that the error analysis of Strassen's algorithm is not based on the standard inner product–based analysis involving the γ_n constant, so that Theorem 3.1 is not directly applicable. Deriving probabilistic error bounds for fast matrix multiplication would not only provide a better understanding of its numerical behavior, but could also help improve its accuracy. Indeed, Dumas et al. [122, 123] develop more accurate variants by minimizing the growth factor appearing in the worst-case bounds; if more descriptive probabilistic bounds can be obtained, minimizing those might lead to even more accurate variants in practice.

16.2.2 Multiword arithmetic

Research Problem 16.3 (Multiword arithmetic for iterative refinement)**.** In Section 5.2, we have explained how fp32 arithmetic can be emulated using multiword arithmetic based on fp16 (or bfloat16) GPU tensor cores. Using the Ozaki approach [210], fp64 arithmetic can also be emulated, usually using int8 arithmetic, but requires a lot more flops. In the context of the solution of linear systems, an interesting problem is how to best combine multiword arithmetic with iterative refinement (Chapter 10). For example, if fp64 accuracy is needed, the following approaches should be compared:

- compute a 16-bit LU factorization and refine the solution to 64-bit accuracy with LU-IR or GMRES-IR (refinement only, no emulation);
- compute a 32-bit LU factorization with fp16-based fp32 emulation and refine the solution to 64-bit accuracy (fp16-based emulation and refinement);
- compute a p-bit LU factorization with int8-based emulation, where $32 \ll p \ll 64$, and refine the solution to 64-bit accuracy (int8-based emulation and refinement);
- compute a 64-bit LU factorization with int8-based fp64 emulation and directly obtain the solution (int8-based emulation only, no refinement).

The best choice between these approaches is likely to depend on both the condition number of the system and the architecture (more specifically, the speed ratio between low and high precision), therefore leading to multiple possible tradeoffs.

Research Problem 16.4 (Multiword arithmetic for randomized low-rank approximations)**.** As explained in Section 6.4, multiword matrix multiplication can be used to accelerate randomized low-rank approximation algorithms. An interesting perspective is to combine this idea with the adaptive precision randomized algorithms described in Section 6.3. Indeed, multiword approaches like the Ozaki scheme [210] can emulate higher precision with a relatively flexible choice of accuracy, depending on the number of words (or "slices") used. Therefore one could accelerate randomized low-rank approximations by progressively reducing the number of words based on an adaptive precision criterion, such as the one derived in Theorem 6.2.

Research Problem 16.5 (Mixed precision multiword arithmetic over prime finite fields)**.** In Section 5.3, we have described a multiword approach to compute $C = AB \bmod p$. A promising perspective is to extend this approach to use mixed precision, and in particular GPU tensor cores that provide 32-bit precision accumulators. Indeed, in this context, we can decompose $A = \sum_{i=0}^{w_A-1} A_i$ and $B = \sum_{j=0}^{w_B-1} B_j$ such that the A_i and B_j matrices are stored in low precision, and we can then efficiently compute $A_iB_j \bmod p$ by blocks, with a large block size λ thanks the high precision accumulation. Interestingly, while storing the matrices in fp16 would in principle allow for integer coefficients of up to 11 bits, the 24-bit size of the fp32 accumulator will in practice become limiting in order to keep the block size λ large enough. For example, if we wish to have $\lambda \geq 2^8$, the integer coefficients of A_i and B_j must fit on 8 bits. This suggests that using int8 arithmetic will in practice be more efficient, since it is twice faster than fp16 arithmetic, and the accumulation uses int32 arithmetic with 32 full bits instead of 24. Moreover, high precision accumulation can also make multimodular arithmetic approaches [114] more efficient, and even superior to our multiword approach. A thorough assessment depending on the storage and accumulation bitwidths should be performed.

16.2.3 Low-rank approximations

Research Problem 16.6 (Robust and efficient fixed-accuracy randomized sketching in finite precision)**.** In Section 6.2, we have described fixed-rank and fixed-accuracy randomized range finder algorithms. While the fixed-accuracy variant is often of greater applicability since it does not require the a priori knowledge of the rank, it is also significantly more expensive. For example, the fixed-accuracy Algorithm 6.4 by Martinsson and Voronin [196] costs $6mnk+o(mnk)$, whereas the fixed-rank Algorithm 6.3 only costs $4mnk + o(mnk)$. This cost difference is further exacerbated when using a sketch that allows for fast matrix–vector products such as subsampled Fourier or Hadamard transforms [154, sect. 4.6], in which case the fixed-accuracy variant can require up to twice as many flops as the fixed-rank one. As mentioned in Section 6.2, it is possible to reduce the cost of the fixed-accuracy variant to almost the same as the fixed-rank one by using a different stopping criterion; rather than explicitly forming the error matrix, the idea is to estimate its norm by implicitly multiplying it with the next sketch [154, sect. 4.4]. However, this criterion is less robust, especially in finite precision arithmetic when the truncation threshold ε approaches the unit roundoff u. An interesting perspective is therefore to investigate the reason behind this numerical issue, and attempt to stabilize the algorithm.

Research Problem 16.7 (Fixed-accuracy randomized interpolative decomposition)**.** A particularly cost-efficient randomized algorithm for computing low-rank approximations is the interpolative decomposition [154, sect. 5.2]. Its bottleneck is indeed the computation of the sketched matrix $A\Omega$, which thus only requires $2mnk+o(mnk)$ flops for a Gaussian matrix Ω, and this cost can even be reduced to $O(mn\log k)$ with subsampled Fourier or Hadamard transforms. However, the literature on randomized interpolative decomposition has mostly focused on the fixed-rank case. A fixed-accuracy variant is quickly mentioned in [187], and we have implemented it in **[37]**, but it is based on the stopping criterion mentioned in the previous Research Problem 16.6, which is not robust. Developing a fixed-accuracy randomized interpolative decomposition based on a robust stopping criterion is an interesting perspective.

Research Problem 16.8 (Quantization of rank-r matrices)**.** In Section 6.7 we developed an algorithm to optimally quantize a rank-one matrix to low precision floating-point arithmetic. Extending this method to rank-r matrices, with $r > 1$, is an interesting perspective. A naive approach would simply quantize each rank-one component separately, but this is clearly not optimal. A better approach would for example consist in quantizing the first rank-one component, deflating it from the matrix, and quantizing the first rank-one component of the resulting deflated matrix, continuing in this manner until r rank-one components have been computed. This iterative approach is self-correcting in the sense that the quantization error for a given component is taken into account when computing the next components. While potentially more accurate, this approach is still not optimal. Whether a provably optimal algorithm can be developed remains an open problem. Moreover, another interesting question is whether we should use the same budget of bits for quantizing each component: clearly, the adaptive precision low-rank approximation described in Section 6.3 suggests that this budget should be progressively decreased.

16.2.4 Krylov solvers

Research Problem 16.9 (High performance mixed precision preconditioned GMRES)**.** The error analysis developed in Section 8.3.1 leads to bounds on the attainable accuracy of preconditioned GMRES that suggest many meaningful mixed precision strategies. This theoretical study should now be complemented with a practical one based on a high performance, parallel

implementation of mixed precision GMRES. This is required to assess the relevant performance–accuracy tradeoffs between different variants on large scale, real-life problems.

Research Problem 16.10 (Mixed precision restarted GMRES with memory accessors)**.** One mixed precision GMRES variant that we have particularly discussed in Section 8.3.1 and that is quite popular among practitioners is to apply the preconditioner in lower precision and recover stability by using flexible GMRES [68, 166, 86]. This, however, comes at the cost of doubling the storage cost for the Krylov basis, which may limit the restart size and thus the convergence rate of the solver. An alternative approach is to rely instead on memory accessors (Chapter 12) to apply the preconditioner in high precision while storing it in low precision; we have seen in Sections 12.3 and 12.4 that LU triangular solves can be performed at the speed of the storage precision. These two competing approaches should be compared as part of a large scale, high performance study on real-life problems.

Research Problem 16.11 (Mixed precision augmented GMRES)**.** Augmented GMRES methods [95, 102] consist in augmenting the standard Krylov subspace with additional vectors in order to speed up the convergence of the solver. In particular, a common approach is to include approximate eigenvectors associated with small eigenvalues [201, 202], to deflate them and make the problem easier to solve. A promising perspective to improve the performance of Krylov solvers is to combine these augmentation techniques with mixed precision, in order to cumulate the reduced number of iterations of the former with the reduced per-iteration cost of the latter. A first attempt was proposed by Oktay and Carson [209], but their method only uses low precision for computing an LU factorization, which is then used as preconditioner for a GCRO-DR [221] solver entirely in high precision. In ongoing work **[46]**, we have indeed observed that GCRO-DR and the closely related GMRES-DR [202] present very limited mixed precision opportunities, because they are based on some algebraic simplifications that only hold approximately in finite precision arithmetic. To overcome this issue, we show that using a more general augmented GMRES method that does not operate these simplifications can leverage lower precision in a much larger part of the solver—possibly all the inner operations including the sparse matrix–vector product, the application of the preconditioner, and the orthonormalization.

16.2.5 BLR solvers

Research Problem 16.12 (Composite data sparse solvers)**.** As discussed in Section 9.5, there exists several data sparse matrix formats: BLR, hierarchical or multilevel formats (such as $\mathcal{H}$ or MBLR), formats with shared or nested bases (such as $\mathcal{H}^2$ or BLR^2), etc. Since the best choice of format depends on various factors such as the problem class or the matrix size, it seems interesting to combine different formats within the same solver; this is particularly natural in sparse direct solvers that require factorizing many dense matrices of various sizes across the levels of the elimination tree (see Section 7.2). Claus et al. [98] describe a composite method that uses ZFP, BLR, and HODBF compression on the smallest, medium, and largest fronts, respectively. This approach can be improved and generalized in three ways. First, it would be useful to determine analytical criterions to decide when to switch from one format to the other; this requires a non-asymptotic complexity analysis of the different formats. Second, rather than switching directly from a fully flat to a fully hierarchical format, using the MBLR format would allow for gradually increasing the number of levels ℓ as we go up in the elimination tree. Third, different formats may also be combined within the same level and even within the same front: for example, using shared/nested bases only for certain parts of the front.

Research Problem 16.13 (Adaptive precision BLR^2 and MBLR formats)**.** The adaptive precision BLR approach described in Section 9.4 should be extended to other data sparse formats.

For MBLR and $\mathcal{H}$ formats, this is natural since each block is compressed independently; Kriemann [176, 177] develops an adaptive precision $\mathcal{H}$-format, although it is mostly used as a compression or storage format; how to extend the adaptive precision BLR LU factorization (which uses lower precisions also in the computations) to multilevel or hierarchical factorizations remains an open question. The case of formats with shared/nested bases, such as BLR^2, is even more difficult. One can expect the coupling matrices to also be representable in adaptive precision form, but the question of whether this is also the case for the shared bases (which can represent a significant part of the total storage) should be investigated. The answer will mainly depend on whether the shared bases exhibit rapidly decaying singular values.

Research Problem 16.14 (Memory accessor for BLR "compute-bound" operations)**.** In Section 12.4, we have described a memory accessor approach to efficiently perform triangular solves with adaptive precision BLR LU factors; in particular, we have shown that memory accessors can handle custom floating-point formats efficiently, which allows for optimizing storage. An important question is whether this memory accessor approach can also be successful for BLR operations traditionally considered to be "compute-bound", such as triangular solves with many right-hand sides (RHS) or LU factorization. While these operations are indeed compute-bound in absence of BLR compression, the use of low-rank approximations strongly reduce their arithmetic intensity, as discussed in Section 9.3. For example, the communication-avoiding approach for BLR triangular solve with many RHS described in that section could be extended to adaptive precision with memory accessors. Similarly, the adaptive precision BLR LU factorization described in Section 9.4 should also be implemented; using a memory accessor is appealing because it would significantly reduce the programming effort by only relying on conversions and not modifying the core computational kernels.

Research Problem 16.15 (BLR sparse solution for FEM-BEM problems)**.** FEM-BEM problems typically require the solution of a linear system involving a matrix $\begin{bmatrix} A & B \\ B^T & C \end{bmatrix}$ where A is symmetric and sparse, B is sparse, and C is dense. How to best tackle these problems with a direct solver employing data sparse formats is an interesting question; a first study was carried out by Agullo et al. [55], but was limited by API constraints of the solver. A first approach to compute the Schur complement $S = C - B^T A^{-1} B$ is to factorize $A = LL^T$ (with BLR compression) and to compute $Y = L^{-1}B$ with forward solves exploiting the sparsity of the right-hand side B; this then yields $S = C - Y^T Y$. Another approach computes the fully dense $Z = L^{-T}L^{-1}B$ with forward and backward solves and then computes $S = C - B^T Z$. The best approach will depend on the sparsity of B and Y, and on the cost of the solves, which also strongly depend on the sparsity and compression format used, as discussed in Section 9.5.5. Moreover, it is also desirable to compress S, and forming the fully dense S (or C) before compressing it is not affordable. A simple approach would be to form S one block at a time. However, a better approach would be to use a randomized compression algorithm based on matrix–vector products with C and with $Y^T Y$ (or $B^T Z$) without ever forming these matrices.

Research Problem 16.16 (Domain decomposition BLR preconditioners)**.** As already mentioned in Research Problem 16.20, it is desirable to reduce the time and memory costs of the local solvers in domain decomposition preconditioners. One idea is to use BLR compression for the subdomains; however, this poses several challenges and questions. First, BLR compression usually requires large or at least medium matrices, but the typical usage in domain decomposition preconditioning is to use small subdomains to reduce the complexity and obtain embarrassing parallelism. In order to benefit from BLR compression, larger subdomains should be used, and parallel local factorizations employed; an additional benefit of increasing the subdomain size

would be to reduce the work of any coarse solver, as well as its criticality for convergence. If the main goal is to reduce the memory consumption of the preconditioner, simply using BLR local solvers might not be sufficient; this is because the memory consumption of BLR solvers is largely dominated by the temporary workspaces needed for the factorization; the size of the compressed LU factors is typically smaller. This represents an issue because domain decomposition preconditioners typically factorize all subdomains in parallel, and so the memory peaks of each local solver will be summed. An interesting idea is to partially sequentialize the subdomains by factorizing them in batches. This approach, combined with a larger subdomain size, however puts more pressure on the scalability of the direct solver.

16.2.6 Adaptive precision algorithms

Research Problem 16.17 (Adaptive precision SpMV formats)**.** As discussed in Section 12.2, the performance of the SpMV strongly depends on the sparse matrix format. When storing the matrix values in low precision, larger speedups are obtained by using a format that minimizes the weight of the indices, since they do not benefit from the use of low precision [203]. A promising perspective is to use such matrix formats for accelerating the adaptive precision SpMV described in Section 11.1. However, this poses significant constraints on the precisions chosen for each matrix element: for example, block formats would require all elements in the same block to be stored in the same precision; ELL-style formats would require for each row, the same number of elements to be stored in each precision.

Research Problem 16.18 (Adaptive precision relaxed GMRES)**.** As explained in Section 8.3.3, it is possible to gradually lower the precision of the matrix–vector product in GMRES as the norm of residual gets smaller. It is timely to revisit relaxed GMRES solvers in light of two recent developments. First, the emergence of many different precision formats supported in hardware, combined with the efficient handling of custom formats thanks to memory accessors (Sections 12.1 and 12.2) makes it possible to refresh the SpMV precision much more frequently. Second, the the adaptive precision SpMV (Section 11.1) allows for controlling its accuracy ε in a continuous matter; while in principle it could thus be refreshed at each iteration, in practice the overhead cost of changing the adaptive precision representation should be taken into account.

Research Problem 16.19 (Adaptive precision incomplete factorizations)**.** Incomplete factorizations such as ILU(k) or ILUT(ε) compute the LU factorization of a sparse matrix while dropping some coefficients of the LU factors in order to enforce a greater level of sparsity than what would be obtained by an exact LU factorization. With such approaches, coefficients are thus either fully dropped or kept in full precision. It seems natural to extend the adaptive precision SpMV from Section 11.1 to incomplete factorizations, by storing the coefficients of the LU factors in progressively lower precisions. The criterion for choosing the precisions could here too be based on the coefficient magnitudes (this would be natural for ILUT(ε) factorizations), or on other criterions such as the sparsity patterns of A, A^2, ..., A^k for ILU(k).

Research Problem 16.20 (Adaptive precision domain decomposition preconditioners)**.** Domain decomposition methods rely on partitioning large sparse linear systems into several much smaller, independent ones, and are thus particularly suited for massively parallel architectures. Nevertheless, for extreme scale problems, the combined cost (in storage and time) of all the local problems can become unaffordable when using exact local solvers. Hence, a promising research direction is the use of approximate methods as local solvers. In this context, a natural question is whether each local subdomain requires the same precision or if, on the contrary, it is meaningful to adapt the precision to each subdomain. In ongoing work **[48]**, we have carried out a

perturbation analysis of the additive Schwarz preconditioner with GenEO coarse sparce [113], and obtained error bounds proportional to $\max_i \kappa(A_i)\varepsilon_i$, where A_i is the matrix associated with the ith subdomain and ε_i is the error incurred in processing A_i; this includes both the factorization of A_i to apply its inverse, and the generalized eigenvalue problem to compute the part of the coarse space associated with A_i. Our error bounds suggest that the precision used in each subdomain should be set to be inversely proportional to its condition number, thereby creating a significant opportunity to use mixed precision.

16.2.7 Tensors

Research Problem 16.21 (Stability of high order TTN computations)**.** Our analysis of the propagation of errors in TTN computations in Section 14.1 produced a bound proportional to α^ℓ, where α is the normalization factor (see Definition 14.1) and ℓ is the number of leaves of the TTN. This bound suggests a possible exponential error growth, and thus a risk of instability, when $\alpha > 1$ and ℓ is large. In practice, however, no such error growth has been observed. Therefore, future work should seek to, on the one hand, find adversarial examples where this bound is attained to understand under which conditions instabilities may arise and, on the other hand, provide an explanation as to why stability is maintained in practice. In particular, Definition 14.1 defines α as the largest possible amplification factor of the error under multiplication. In practice, the actual amplification factors may be smaller, especially if the error matrix is structured. An important example is the sum of two e-normalized TTNs. In this case, the 1-normalized, nonleaf nodes have a block diagonal structure. If the error produced on these nodes retains their block diagonal structure, then multiplying it by a $\sqrt{2}$-normalized leaf node does not amplify the error at all, since each diagonal block is multiplied by an orthonormal matrix. This observation should be formalized to rigorously establish in which cases stability is provably maintained.

Research Problem 16.22 (Performance comparison of TTN rounding algorithms)**.** Two approaches to perform TTN rounding should be compared. On the one hand, Gram LRA–based approaches [216, 141, 175, 58] are very parallel but unstable, preventing the use of lower precision arithmetic. On the other hand, orthonormalization-based approaches [215, 243], **[39]** are less parallel, because they require truncating each dimension sequentially, but stable, and can thus safely exploit lower precision. This motivates a pratical study comparing the performance of these two types of approaches on various computer architectures depending on their parallel and arithmetic environment. Moreover, they should also be compared with other rounding methods, such as randomized ones [57]. Finally, one may wonder to what extent it is necessary to transfer the non-orthonormality to the node being rounded. Indeed, our error analysis in Section 14.1 shows that, to ensure stability, it is sufficient that all nodes except one are orthonormal, but that one node need not be the one being rounded. While this would allow to parallelize the method, rounding orthonormal nodes may result in suboptimal dimensions.

Research Problem 16.23 (Adaptive precision tensor decompositions)**.** It is natural to seek to extend the adaptive precision low-rank matrix representation described in Section 6.3 to higher order tensors. One difficulty is that this representation is based on the singular values of the matrix, but tensors do not have a direct equivalent. One idea could be to employ this adaptive precision representation for the underlying low-rank matrix approximations that are computed on matricized tensors. Another approach could instead be to generalize the idea of looking at the singular values by taking into account the magnitudes of other coefficients in the low-rank tensor that play a similar role; the so-called "core" node of Tucker tensors seems a natural candidate.

Research Problem 16.24 (Quantization of low-rank tensors)**.** In Section 6.7, we have shown how to optimally quantize a rank-one matrix xy^T to t-bit floating-point arithmetic. Importantly,

the optimal error can be much smaller than that achieved by naively quantizing x and y individually. It seems likely that this observation continues to hold for rank-one tensors of the form $x_1 \otimes \cdots \otimes x_d$, whose entries are represented as the product of d t-bit floating-point numbers. Finding an optimal quantization algorithm, and whether one of tractable complexity exists, is an open problem.

16.2.8 Neural networks

Research Problem 16.25 (Adaptive precision pruning for neural networks)**.** A common method to compress large neural networks is to sparsify their weights by removing connections, also known as pruning [77]. One approach consists in pruning weights of small magnitude [157]. While such pruning can be naturally combined with low precision quantization by quantizing the weights that are not pruned, a more efficient approach could be to employ magnitude-based adaptive precision quantization, as in the SpMV approach developed in Section 11.1.

Research Problem 16.26 (Adaptive precision low-rank approximations for neural networks)**.** Low-rank approximations are also commonly employed for compressing large neural networks, such as in the LoRA algorithm [167]. While LoRA has been combined with low precision quantization [112, 254], a more efficient approach could be to employ an adaptive precision quantization based on the approach developed in Section 6.3. It remains to be seen whether the low-rank structures in these networks exhibit a sufficient singular value decay for the use of multiple precisions to be meaningful.

Research Problem 16.27 (Quantized butterfly factorizations for neural networks)**.** As discussed in Section 13.2, butterfly matrices have been used to compress neural networks [105, 106, 103, 104]. In Section 13.3, we have developed an approach to quantize butterfly matrices to low precision, based on our rank-one matrix optimal quantization described in Section 6.7. It seems promising to explore the use of this quantized butterfly factorization to compress neural networks.

Research Problem 16.28 (Static mixed precision accumulation)**.** In Section 15.2, we have proposed a mixed precision algorithm for neural network inference that accumulates different inner products in different precisions. The algorithm is dynamic in the sense that we first compute all inner products in low precision and then dynamically decide which ones to recompute in high precision based on their estimated condition number. While this approach is robust, the recompute overhead cost makes it practical only if a small fraction of inner products need to be (re)computed in high precision. This overhead cost could be removed by instead statically deciding which inner products to compute a priori in high precision—regardless of the input. Naturally, this requires "learning" the precision configuration on a representative set of input vectors. This static strategy may achieve better performance–accuracy tradeoffs than the dynamic strategy presented in Section 15.2. Moreover, using a static precision configuration would allow for exploiting mixed precision not just for the accumulation, but for the quantization too.

Research Problem 16.29 (Error analysis for transformers)**.** The error analyses developed in Section 15.1 and Section 15.2 consider MLP networks with fully-connected layers. An interesting perspective is to extend these analyses to other types of network architectures, in particular transformers [244]; they have been analyzed by Budzinskiy et al. [80], but further investigation is required, in particular, concerning mixed precision opportunities. Indeed, transformers involve the softmax function, which amplifies variations in magnitude in the data—the mixed precision approach developed in Section 15.2 might thus be applicable.

References: personal publications

Journal articles

[1] N. J. Higham and T. Mary. A new approach to probabilistic rounding error analysis. *SIAM J. Sci. Comput.* 41.5 (2019), A2815–A2835.

[2] N. J. Higham and T. Mary. Sharper probabilistic backward error analysis for basic linear algebra kernels with random data. *SIAM J. Sci. Comput.* 42.5 (2020), A3427–A3446.

[3] M. P. Connolly, N. J. Higham, and T. Mary. Stochastic rounding and its probabilistic backward error analysis. *SIAM J. Sci. Comput.* 43.1 (2021), A566–A585.

[4] M. Croci, M. Fasi, N. J. Higham, T. Mary, and M. Mikaitis. Stochastic rounding: Implementation, error analysis and applications. *Roy. Soc. Open Sci.* 9.3 (2022), 1–25.

[5] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh. Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores. *SIAM J. Sci. Comput.* 42.3 (2020), C124–C141.

[6] T. Mary and M. Mikaitis. Error analysis of matrix multiplication with narrow range floating-point arithmetic. *SIAM J. Sci. Comput.* 47.4 (2025), B785–B800.

[7] M. Fasi, N. J. Higham, F. Lopez, T. Mary, and M. Mikaitis. Matrix multiplication in multiword arithmetic: Error analysis and application to GPU tensor cores. *SIAM J. Sci. Comput.* (2023).

[8] N. J. Higham and T. Mary. Solving block low-rank linear systems by LU factorization is numerically stable. *IMA J. Numer. Anal.* 42.2 (2021), 951–980.

[9] T. Mary. Error analysis of the Gram low-rank approximation (and why it is not as unstable as one may think). *SIAM J. Matrix Anal. Appl.* 46.2 (2025), 1444–1459.

[10] N. J. Higham and T. Mary. Mixed precision algorithms in numerical linear algebra. *Acta Numerica* 31 (2022), 347–414.

[11] P. R. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieublé. Five-precision GMRES-based iterative refinement. *SIAM J. Matrix Anal. Appl.* 45.1 (2024), 529–552.

[12] A. Buttari, N. J. Higham, T. Mary, and B. Vieublé. A modular framework for the backward error analysis of GMRES. HAL EPrint hal-04525918; to appear in *IMA J. Numer. Anal.*.

[13] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Trans. Math. Software* 45.1 (2019), 2:1–2:26.

[14] P. R. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary. Communication avoiding block low-rank parallel multifrontal triangular solve with many right-hand sides. *SIAM J. Matrix Anal. Appl.* 45.1 (2024), 148–166.

[15] P. R. Amestoy, A. Buttari, N. J. Higham, J.-Y. L'Excellent, T. Mary, and B. Vieublé. Combining sparse approximate factorizations with mixed precision iterative refinement. *ACM Trans. Math. Software* 49.1 (2023).

[16] P. R. Amestoy, O. Boiteau, A. Buttari, M. Gerest, F. Jézéquel, J.-Y. L'Excellent, and T. Mary. Mixed precision low rank approximations and their application to block low rank LU factorization. *IMA J. Numer. Anal.* 43.4 (2022), 2198–2227.

[17] N. J. Higham and T. Mary. A new preconditioner that exploits low-rank approximations to factorization error. *SIAM J. Sci. Comput.* 41.1 (2019), A59–A82.

[18] S. Graillat, F. Jézéquel, T. Mary, and R. Molina. Adaptive precision sparse matrix-vector product and its application to Krylov solvers. *SIAM J. Sci. Comput.* 46.1 (2024), C30–C56.

[19] P. Blanchard, N. J. Higham, and T. Mary. A class of fast and accurate summation algorithms. *SIAM J. Sci. Comput.* 42.3 (2020), A1541–1557.

[20] S. Graillat and T. Mary. Condense and distill: fast distillation of large floating-point sums via condensation. *SIAM J. Sci. Comput.* 47.2 (2025), B583–B594.

[21] A. Buttari, T. Mary, and A. Pacteau. Truncated QR factorization with pivoting in mixed precision. *SIAM J. Sci. Comput.* 47.2 (2025), B382–B401.

[22] M. Baboulin, O. Kaya, T. Mary, and M. Robeyns. Mixed precision iterative refinement for low-rank matrix and tensor approximations. HAL EPrint hal-04115337; to appear in *SIAM J. Sci. Comput.*.

[23] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. On the complexity of the block low-rank multifrontal factorization. *SIAM J. Sci. Comput.* 39.4 (2017), A1710–A1740.

[24] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Bridging the gap between flat and hierarchical low-rank matrix formats: The multilevel block low-rank format. *SIAM J. Sci. Comput.* 41.3 (2019), A1414–A1442.

[25] C. Ashcraft, A. Buttari, and T. Mary. Block low-rank matrices with shared bases: Potential and limitations of the BLR^2 format. *SIAM J. Matrix Anal. Appl.* 42.2 (2021), 990–1010.

[26] C.-P. Jeannerod, T. Mary, C. Pernet, and D. S. Roche. Improving the complexity of block low-rank factorizations with fast matrix arithmetic. *SIAM J. Matrix Anal. Appl.* 40.4 (2019), 1478–1496.

[27] F. Lopez and T. Mary. Mixed precision LU factorization on GPU tensor cores: reducing data movement and memory footprint. *Int. J. High Perform. Comput. Appl.* 37.2 (2023), 165–179.

[28] S. Operto, P. R. Amestoy, H. Aghamiry, S. Beller, A. Buttari, L. Combe, V. Dolean, M. Gerest, G. Guo, P. Jolivet, J.-Y. L'Excellent, F. Mamfoumbi, T. Mary, C. Puglisi, A. Ribodetti, and P.-H. Tournier. Is 3D frequency-domain FWI of full-azimuth/long-offset OBN data feasible? The Gorgon case study. *Leading Edge* 42.3 (2023), 146–228.

[29] A. Buttari, M. Huber, P. Leleux, T. Mary, U. Ruede, and B. Wohlmuth. Block low rank single precision coarse grid solvers for extreme scale multigrid methods. *Numer. Linear Algebra Appl.* 29.1 (2021).

[30] C. Gorman, G. Chavez, P. Ghysels, T. Mary, F.-H. Rouet, and X. S. Li. Robust and accurate stopping criteria for adaptive randomized sampling in matrix-free hierarchically semiseparable construction. *SIAM J. Sci. Comput.* 41.5 (2019), S61–S85.

[31] P. R. Amestoy, R. Brossier, A. Buttari, J.-Y. L'Excellent, T. Mary, L. Métivier, A. Miniussi, and S. Operto. Fast 3D frequency-domain full waveform inversion with a parallel block low-rank multifrontal direct solver: Application to OBC data from the North Sea. *Geophysics* 81.6 (2016), R363–R383.

[32] D. V. Shantsev, P. Jaysaval, S. de la Kethulle de Ryhove, P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Large-scale 3D EM modeling with a block low-rank multifrontal direct solver. *Geophys. J. Int.* 209.3 (2017), 1558–1571.

Conference articles

[33] M. Baboulin, S. Donfack, O. Kaya, T. Mary, and M. Robeyns. Mixed precision randomized low-rank approximation with GPU tensor cores. *Euro-Par 2024: Parallel Processing*. Cham: Springer Nature Switzerland, 2024, 31–44.

[34] P. R. Amestoy, A. Buttari, F. Faucher, J.-Y. L'Excellent, and T. Mary. Recent work on sparse direct solvers to exploit numerical and structural properties of HDG discretizations. *14th WCCM and ECCOMAS congress 2020, Paris, France*. Jan. 2021.

[35] P. R. Amestoy, A. Buttari, F. Faucher, M. Gerest, J.-Y. L'Excellent, and T. Mary. Mixed precision sparse direct solver applied to 3D wave propagation. *ECCOMAS congress 2022, Oslo, Norway*. June 2022.

[36] S. Graillat, F. Jézéquel, T. Mary, R. Molina, and D. Mukunoki. Reduced-precision and reduced-exponent formats for accelerating adaptive precision sparse matrix-vector product. *Euro-Par 2024: Parallel Processing*. Cham: Springer Nature Switzerland, 2024, 17–30.

[37] T. Mary, I. Yamazaki, J. Kurzak, P. Luszczek, S. Tomov, and J. Dongarra. Performance of random sampling for computing low-rank approximations of a dense matrix on GPUs. *SC'15 - International Conference for High Performance Computing, Networking, Storage and Analysis*. Austin, USA, Nov. 2015.

Preprints

[38] F. Jézéquel and T. Mary. Probabilistic estimation of the accuracy of inner products and application to stochastic validation. HAL EPrint hal-04554459.

[39] M. Baboulin, O. Kaya, T. Mary, and M. Robeyns. Numerical stability of tree tensor network operations, and a stable rounding algorithm. HAL EPrint hal-04996127.

[40] A. Buttari, X. Liu, T. Mary, and B. Vieublé. Mixed precision strategies for preconditioned GMRES: a comprehensive analysis. HAL EPrint hal-05071696.

[41] P. R. Amestoy, A. Jego, J.-Y. L'Excellent, T. Mary, and G. Pichon. BLAS-based block memory accessor with applications to mixed precision sparse direct solvers. HAL EPrint hal-05019106.

[42] R. Gribonval, T. Mary, and E. Riccietti. Optimal quantization of rank-one matrices in floating-point arithmetic—with applications to butterfly factorizations. HAL EPrint hal-04125381.

[43] E.-M. E. Arar, S. Filip, T. Mary, and E. Riccietti. Mixed precision accumulation for neural network inference guided by componentwise forward error analysis. HAL EPrint hal-04995708.

[44] T. Beuzeville, A. Buttari, S. Gratton, and T. Mary. Deterministic and probabilistic backward error analysis of neural networks in floating-point arithmetic. HAL EPrint hal-04663142.

[45] J. Berthomieu, S. Graillat, D. Lesnoff, and T. Mary. Multiword matrix multiplication over large finite fields in floating-point arithmetic. HAL EPrint hal-04917201.

[46] Y. Jang, P. Jolivet, and T. Mary. Mixed precision augmented GMRES. HAL EPrint hal-05163845.

In preparation

[47] A. Anciaux-Sedrakian, H. Dorfsman, T. Guignon, F. Jézéquel, and T. Mary. Mixed precision strategies for solving sparse linear systems with BiCGStab. In preparation.

[48] T. Caruso, P. Jolivet, T. Mary, F. Nataf, and P.-H. Tournier. Perturbation analysis of the mixed precision additive Schwarz preconditioner with GenEO coarse space. In preparation.

PhD thesis

[49] T. Mary. Block Low-Rank multifrontal solvers: complexity, performance, and scalability. PhD Thesis. Université de Toulouse, Nov. 2017.

Unpublished

[50] P. Amestoy, A. Buttari, J.-Y. L'Excellent, T. Mary, and G. Moreau. Asymptotic complexity of low-rank sparse direct solvers with sparse right-hand sides. *SIAM Conference on Combinatorial Scientific Computing (SIAM CSC'20), Seattle, USA*. Feb. 2020.

[51] T. Beuzeville, A. Buttari, S. Gratton, T. Mary, and S. Pralet. Adversarial attacks via backward error analysis. HAL EPrint hal-03296180. 2021.

[52] T. Beuzeville, A. Buttari, S. Gratton, T. Mary, and E. Ulker. Adversarial attacks via sequential quadratic programming. HAL EPrint hal-03752184. 2022.

Other references

[53] S. Abdulah, Q. Cao, Y. Pei, G. Bosilca, J. Dongarra, M. G. Genton, D. E. Keyes, H. Ltaief, and Y. Sun. Accelerating geostatistical modeling and prediction with mixed-precision computations: a high-productivity approach with PaRSEC. *IEEE Trans. Parallel Distrib. Syst.* 33.4 (2022), 964–976.

[54] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, and D. E. Keyes. Geostatistical modeling and prediction using mixed precision tile Cholesky factorization. *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, Dec. 2019.

[55] E. Agullo, M. Felšöci, and G. Sylvand. Direct solution of larger coupled sparse/dense linear systems using low-rank compression on single-node multi-core machines in an industrial context. *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2022, 25–35.

[56] S. Ahmadi-Asl, S. Abukhovich, M. G. Asante-Mensah, A. Cichocki, A. H. Phan, T. Tanaka, and I. Oseledets. Randomized algorithms for computation of Tucker decomposition and higher order SVD (HOSVD). *IEEE Access* 9 (2021), 28684–28706.

[57] H. Al Daas, G. Ballard, P. Cazeaux, E. Hallman, A. Międlar, M. Pasha, T. W. Reid, and A. K. Saibaba. Randomized algorithms for rounding in the tensor-train format. *SIAM J. Sci. Comput.* 45.1 (2023), A74–A95.

[58] H. Al Daas, G. Ballard, and L. Manning. Parallel tensor train rounding using Gram SVD. *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2022, 930–940.

[59] J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. E. Tomás. Compressed basis GMRES on high-performance graphics processing units. *Int. J. High Perform. Comput. Appl.* 37.2 (2023), 82–100.

[60] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.* 23.1 (2001), 15–41.

[61] P. R. Amestoy, S. de la Kethulle de Ryhove, J.-Y. L'Excellent, G. Moreau, and D. V. Shantsev. Efficient use of sparsity by direct solvers applied to 3D controlled-source EM problems. *Comput. Geosci.* 23 (Sept. 2019), 1237–1258.

[62] P. R. Amestoy, J.-Y. L'Excellent, and G. Moreau. On exploiting sparsity of multiple right-hand sides in sparse direct solvers. *SIAM J. Sci. Comput.* 41.1 (2019), A269–A291.

[63] A. Aminfar, S. Ambikasaran, and E. Darve. A fast block low-rank dense solver with applications to finite-element matrices. *J. Comput. Phys.* 304 (2016), 170–188.

[64] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang. Load-balancing sparse matrix vector product kernels on GPUs. *ACM Trans. Parallel Comput.* 7 (Mar. 2020).

[65] H. Anzt, J. Dongarra, G. Flegar, N. J. Higham, and E. S. Quintana-Ortí. Adaptive precision in block-Jacobi preconditioning for iterative sparse linear system solvers. *Concurr. Comput. Pract. Exp.* 31.6 (2019), e4460.

[66] H. Anzt, G. Flegar, T. Grützmacher, and E. S. Quintana-Ortí. Toward a modular precision ecosystem for high-performance computing. *Int. J. High Perform. Comput. Appl.* 33.6 (2019), 1069–1078.

[67] E.-M. E. Arar, D. Sohier, P. de Oliveira Castro, and E. Petit. Stochastic rounding variance and probabilistic bounds: a new approach. *SIAM J. Sci. Comput.* 45.5 (2023), C255–C275.

[68] M. Arioli and I. S. Duff. Using FGMRES to obtain backward stability in mixed precision. *Electron. Trans. Numer. Anal.* 33 (2009), 31–44.

[69] M. Arioli, I. S. Duff, S. Gratton, and S. Pralet. A note on GMRES preconditioned by a perturbed LDL^T decomposition with static pivoting. *SIAM J. Sci. Comput.* 29.5 (2007), 2024–2044.

[70] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Comput. Phys. Comm.* 180.12 (2009), 2526–2533.

[71] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov. Accelerating linear system solutions using randomization techniques. *ACM Trans. Math. Software* 39.2 (Feb. 2013).

[72] M. Baboulin, X. S. Li, and F.-H. Rouet. Using random butterfly transformations to avoid pivoting in sparse direct methods. *High Performance Computing for Computational Science – VECPAR 2014*. Ed. by M. Daydé, O. Marques, and K. Nakajima. Cham: Springer International Publishing, 2015, 135–144.

[73] O. Balabanov and L. Grigori. Randomized Gram–Schmidt process with application to GMRES. *SIAM J. Sci. Comput.* 44.3 (2022), A1450–A1474.

[74] M. Bebendorf. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*. Vol. 63. Lecture Notes in Computational Science and Engineering (LNCSE). Springer-Verlag, 2008.

[75] M. Bebendorf and W. Hackbusch. Existence of $\mathcal{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with L^∞-coefficients. *Numer. Math.* 95.1 (2003), 1–28.

[76] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. 2008.

[77] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag. What is the state of neural network pruning? *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, 129–146.

[78] S. Börm, L. Grasedyck, and W. Hackbusch. Introduction to hierarchical matrices with applications. *Eng. Anal. Bound. Elem.* 27.5 (2003), 405–422.

[79] A. Bouras and V. Frayssé. Inexact matrix-vector products in Krylov methods for solving linear systems: a relaxation strategy. *SIAM J. Matrix Anal. Appl.* 26.3 (2005), 660–678.

[80] S. Budzinskiy, W. Fang, L. Zeng, and P. Petersen. Numerical error analysis of large language models. arXiv:2503.10251. 2025.

[81] L. Burke, E. Carson, and Y. Ma. On the numerical stability of sketched GMRES. arXiv:2503.19086. 2025.

[82] P. Businger and G. H. Golub. Linear least squares solutions by Householder transformations. *Numer. Math.* 7.3 (June 1965), 269–276.

[83] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Software* 34.4 (July 2008), 17:1–17:22.

[84] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. High Perform. Comput. Appl.* 21 (Nov. 2007).

[85] E. Carson and I. Daužickaitė. Mixed precision sketching for least-squares problems and its application in GMRES-based iterative refinement. arXiv:2410.06319. 2024.

[86] E. Carson and I. Daužickaitė. The stability of split-preconditioned FGMRES in four precisions. *Electron. Trans. Numer. Anal.* 60 (2024), 40–58.

[87] E. Carson and N. J. Higham. A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM J. Sci. Comput.* 39.6 (2017), A2834–A2856.

[88] E. Carson and N. J. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM J. Sci. Comput.* 40.2 (2018), A817–A847.

[89] E. Carson, N. J. Higham, and S. Pranesh. Three-precision GMRES-based iterative refinement for least squares problems. *SIAM J. Sci. Comput.* 42.6 (2020), A4063–A4083.

[90] E. Carson and N. Khan. Mixed precision iterative refinement with sparse approximate inverse preconditioning. *SIAM J. Sci. Comput.* 45.3 (June 2023), C131–C153.

[91] E. Carson, K. Lund, M. Rozložník, and S. Thomas. Block Gram–Schmidt algorithms and their stability properties. *Linear Algebra Appl.* 638 (Apr. 2022), 150–195.

[92] E. Carson and Y. Ma. On the backward stability of s-step GMRES. arXiv:2409.03079. 2024.

[93] A. M. Castaldo, R. C. Whaley, and A. T. Chronopoulos. Reducing floating point error in dot product using the superblock family of algorithms. *SIAM J. Sci. Comput.* 31.2 (2009), 1156–1174.

[94] S. Chandrasekaran, M. Gu, and T. Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM J. Matrix Anal. Appl.* 28.3 (2006), 603–622.

[95] A. Chapman and Y. Saad. Deflated and augmented Krylov subspace techniques. *Numer. Linear Algebra Appl.* 4.1 (1997), 43–66.

[96] B. Chen, T. Dao, E. Winsor, Z. Song, A. Rudra, and C. Ré. Scatterbrain: unifying sparse and low-rank attention. *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan. Vol. 34. Curran Associates, Inc., 2021, 17413–17426.

[97] L. Claus, P. Ghysels, W. H. Boukaram, and X. S. Li. A graphics processing unit accelerated sparse direct solver and preconditioner with block low rank compression. *Int. J. High Perform. Comput. Appl.* 39.1 (2025), 18–31.

[98] L. Claus, P. Ghysels, Y. Liu, T. A. Nhan, R. Thirumalaisamy, A. P. S. Bhalla, and S. Li. Sparse approximate multifrontal factorization with composite compression methods. *ACM Trans. Math. Software* 49.3 (Sept. 2023).

[99] M. P. Connolly and N. J. Higham. Probabilistic rounding error analysis of Householder QR factorization. *SIAM J. Matrix Anal. Appl.* 44.3 (2023), 1146–1163.

[100] M. P. Connolly, N. J. Higham, and S. Pranesh. Randomized low rank matrix approximation: Rounding error analysis and a mixed precision algorithm. MIMS EPrint 2022.5. UK, July 2022.

[101] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Math. Comput.* 19 (1965), 297–301.

[102] O. Coulaud, L. Giraud, P. Ramet, and X. Vasseur. Deflation and augmentation techniques in Krylov subspace methods for the solution of linear systems. arXiv:1303.5692. 2013.

[103] T. Dao, B. Chen, K. Liang, J. Yang, Z. Song, A. Rudra, and C. Ré. Pixelated butterfly: simple and efficient sparse training for neural network models. arXiv:2112.00029. 2022.

[104] T. Dao, B. Chen, N. S. Sohoni, A. Desai, M. Poli, J. Grogan, A. Liu, A. Rao, A. Rudra, and C. Re. Monarch: expressive structured matrices for efficient and accurate training. *Proceedings of the 39th International Conference on Machine Learning*. Ed. by K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato. Vol. 162. Proceedings of Machine Learning Research. PMLR, July 2022, 4690–4721.

[105] T. Dao, A. Gu, M. Eichhorn, A. Rudra, and C. Re. Learning fast algorithms for linear transforms using butterfly factorizations. *Proceedings of the 36th International Conference on Machine Learning*. Ed. by K. Chaudhuri and R. Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, June 2019, 1517–1527.

[106] T. Dao, N. S. Sohoni, A. Gu, M. Eichhorn, A. Blonder, M. Leszczynski, A. Rudra, and C. Ré. Kaleidoscope: an efficient, learnable representation for all structured linear maps. arXiv:2012.14966. 2021.

[107] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl.* 21.4 (2000), 1253–1278.

[108] J. Demmel and Y. Hida. Accurate and efficient floating point summation. *SIAM J. Sci. Comput.* 25.4 (2004), 1214–1248.

[109] J. Demmel, Y. Hida, W. Kahan, X. S. Li, S. Mukherjee, and E. J. Riedy. Error bounds from extra-precise iterative refinement. *ACM Trans. Math. Software* 32.2 (June 2006), 325–351.

[110] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang. Communication avoiding rank revealing QR factorization with column pivoting. *SIAM J. Matrix Anal. Appl.* 36.1 (2015), 55–89.

[111] M. Dessole and F. Marcuzzi. Deviation maximization for rank-revealing QR factorizations. *Numer. Algorithms* 91.3 (Nov. 2022), 1047–1079.

[112] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. QLoRA: efficient finetuning of quantized LLMs. *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., 2023, 10088–10115.

[113] V. Dolean, P. Jolivet, and F. Nataf. *An Introduction to Domain Decomposition Methods*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2015.

[114] J. Doliskani, P. Giorgi, R. Lebreton, and E. Schost. Simultaneous conversions with the residue number system using linear algebra. *ACM Trans. Math. Software* 44.3 (Jan. 2018).

[115] N. Doucet, H. Ltaief, D. Gratadour, and D. Keyes. Mixed-precision tomographic reconstructor computations on hardware accelerators. *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. Nov. 2019, 31–38.

[116] J. Drkošová, A. Greenbaum, M. Rozložník, and Z. Strakoš. Numerical stability of GMRES. *BIT Numer. Math.* 35.3 (1995), 309–330.

[117] J. A. Duersch and M. Gu. Randomized QR with column pivoting. *SIAM J. Sci. Comput.* 39.4 (2017), C263–C291.

[118] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Second. Oxford University Press, 2017, xiii+429.

[119] I. S. Duff and S. Pralet. Towards stable mixed pivoting strategies for the sequential and parallel solution of sparse symmetric indefinite systems. *SIAM J. Matrix Anal. Appl.* 29.3 (2007), 1007–1024.

[120] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Software* 9 (1983), 302–325.

[121] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the FFLAS and FFPACK packages. *ACM Trans. Math. Software* 35.3 (2008), 1–42.

[122] J.-G. Dumas, C. Pernet, and A. Sedoglavic. Strassen's algorithm is not optimally accurate. *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*. ISSAC '24. Raleigh, NC, USA: Association for Computing Machinery, 2024, 254–263.

[123] J.-G. Dumas, C. Pernet, and A. Sedoglavic. Towards automated generation of fast and accurate algorithms for recursive matrix multiplication. HAL EPrint hal-04995684. Mar. 2025.

[124] C. Eckard and G. Young. The approximation of one matrix by another of lower rank. *Psychometrica* 1 (1936), 211–218.

[125] E.-M. El Arar, M. Fasi, S.-I. Filip, and M. Mikaitis. Probabilistic error analysis of limited-precision stochastic rounding. arXiv:2408.03069. 2025.

[126] E.-M. El Arar, D. Sohier, P. de Oliveira Castro, and E. Petit. Bounds on nonlinear errors for variance computation with stochastic rounding. *SIAM J. Sci. Comput.* 46.5 (2024), B579–B599.

[127] J. van den Eshof and G. L. G. Sleijpen. Inexact Krylov subspace methods for linear systems. *SIAM J. Matrix Anal. Appl.* 26.1 (2004), 125–153.

[128] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh. Numerical behavior of NVIDIA tensor cores. *PeerJ Comput. Sci.* 7 (Feb. 2021), e330(1–19).

[129] G. Flegar, H. Anzt, T. Cojean, and E. S. Quintana-Ortí. Adaptive precision block-Jacobi for high performance preconditioning in the Ginkgo linear algebra software. *ACM Trans. Math. Software* 47.2 (Apr. 2021), 1–28.

[130] L. Fox, H. D. Huskey, and J. H. Wilkinson. Notes on the solution of algebraic linear simultaneous equations. *Quart. J. Mech. Appl. Math.* 1 (1948), 149–173.

[131] T. Fukaya, R. Kannan, Y. Nakatsukasa, Y. Yamamoto, and Y. Yanagisawa. Shifted Cholesky QR for computing the QR factorization of ill-conditioned matrices. *SIAM J. Sci. Comput.* 42.1 (2020), A477–A503.

[132] T. Fukaya, Y. Nakatsukasa, and Y. Yamamoto. A Cholesky QR type algorithm for computing tall-skinny QR factorization with column pivoting. *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2024, 63–75.

[133] T. Fukaya, Y. Nakatsukasa, Y. Yanagisawa, and Y. Yamamoto. CholeskyQR2: a simple and communication-avoiding algorithm for computing a tall-skinny QR factorization on a large-scale parallel system. *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 2014, 31–38.

[134] J. E. Garrison and I. C. F. Ipsen. A randomized preconditioned Cholesky-QR algorithm. arXiv:2406.11751. 2024.

[135] J. A. George. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.* 10.2 (1973), 345–363.

[136] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM J. Sci. Comput.* 38.5 (2016), S358–S384.

[137] P. E. Gill, M. A. Saunders, and J. R. Shinnerl. On the stability of Cholesky factorization for symmetric quasidefinite systems. *SIAM J. Matrix Anal. Appl.* 17.1 (1996), 35–46.

[138] A. Gillman, P. Young, and P.-G. Martinsson. A direct solver with $\mathcal{O}(N)$ complexity for integral equations on one-dimensional domains. *Front. Math. China* 7 (2 2012), 217–247.

[139] L. Giraud, S. Gratton, and J. Langou. Convergence in backward error of relaxed GMRES. *SIAM J. Sci. Comput.* 29.2 (2007), 710–728.

[140] L. Giraud, J. Langou, M. Rozložník, and J. van den Eshof. Rounding error analysis of the classical Gram–Schmidt orthogonalization process. *Numer. Math.* 101.1 (May 2005), 87–100.

[141] L. Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM J. Matrix Anal. Appl.* 31.4 (2010), 2029–2054.

[142] L. Grasedyck, D. Kressner, and C. Tobler. A literature survey of low-rank tensor approximation techniques. *GAMM Mitt.* 36.1 (2013), 53–78.

[143] S. Gratton, E. Simon, D. Titley-Peloquin, and P. Toint. Exploiting variable precision in GMRES. arXiv:1907.10550. July 2019.

[144] S. Gratton, E. Simon, D. Titley-Peloquin, and P. L. Toint. A note on inexact inner products in GMRES. *SIAM J. Matrix Anal. Appl.* 43.3 (2022), 1406–1422.

[145] A. Greenbaum, V. Pták, and Z. Strakoš. Any nonincreasing convergence curve is possible for GMRES. *SIAM J. Matrix Anal. Appl.* 17.3 (1996), 465–469.

[146] T. Grützmacher, H. Anzt, and E. S. Quintana-Ortí. Using Ginkgo's memory accessor for improving the accuracy of memory-bound low precision BLAS. *Software—Practice and Experience* (Oct. 2021).

[147] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, 1737–1746.

[148] S. Güttel and I. Simunec. A sketch-and-select Arnoldi process. *SIAM J. Sci. Comput.* 46.4 (2024), A2774–A2797.

[149] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing* 62.2 (1999), 89–108.

[150] W. Hackbusch. *Hierarchical Matrices : Algorithms and Analysis*. Vol. 49. Springer Series in Computational Mathematics. Berlin: Springer, 2015, xxv, 511.

[151] W. Hackbusch and S. Kühn. A new scheme for the tensor representation. *J. Fourier Anal. Appl.* 15.5 (2009), 706–722.

[152] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham. Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems. *Proc. Roy. Soc. London A* 476.2243 (2020), 20200110.

[153] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC'18. Dallas, Texas: IEEE Press, 2018, 47:1–47:11.

[154] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.* 53.2 (2011), 217–288.

[155] E. Hallman. A refined probabilistic error bound for sums. arXiv:2104.06531. 2021.

[156] E. Hallman and I. C. F. Ipsen. Precision-aware deterministic and probabilistic error bounds for floating point summation. *Numer. Math.* 155.1–2 (Aug. 2023), 83–119.

[157] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems*. Ed. by C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett. Vol. 28. Curran Associates, Inc., 2015.

[158] G. Henry, P. T. P. Tang, and A. Heinecke. Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 2019, 69–76.

[159] N. J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *ACM Trans. Math. Software* 16.4 (Dec. 1990), 352–368.

[160] N. J. Higham. Iterative refinement enhances the stability of QR factorization methods for solving linear equations. *BIT* 31 (1991), 447–468.

[161] N. J. Higham. Iterative refinement for linear systems and LAPACK. *IMA J. Numer. Anal.* 17.4 (1997), 495–509.

[162] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Second. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, xxx+680.

[163] N. J. Higham and S. Pranesh. Exploiting lower precision arithmetic in solving symmetric positive definite linear systems and least squares problems. *SIAM J. Sci. Comput.* 43.1 (2021), A258–A277.

[164] N. J. Higham, S. Pranesh, and M. Zounon. Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM J. Sci. Comput.* 41.4 (2019), A2536–A2551.

[165] M. Hoemmen. Communication-avoiding Krylov subspace methods. PhD thesis. University of California, Berkeley, 2010.

[166] J. D. Hogg and J. A. Scott. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Trans. Math. Software* 37.2 (Apr. 2010), 1–24.

[167] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: low-rank adaptation of large language models. *ICLR*. OpenReview.net, 2022.

[168] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Available at `https://cdrdv2.intel.com/v1/dl/getContent/671200`. Intel. June 2024.

[169] I. C. F. Ipsen and H. Zhou. Probabilistic error analysis for inner products. *SIAM J. Matrix Anal. Appl.* 41.4 (Jan. 2020), 1726–1741.

[170] T. Iwashita, K. Suzuki, and T. Fukaya. An integer arithmetic-based sparse linear solver using a GMRES method and iterative refinement. *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 2020, 1–8.

[171] M. Jankowski and H. Woźniakowski. Iterative refinement implies numerical stability. *BIT Numer. Math.* (1977), 303–311.

[172] C.-P. Jeannerod and S. M. Rump. Improved error bounds for inner products in floating-point arithmetic. *SIAM J. Matrix Anal. Appl.* 34.2 (2013), 338–344.

[173] F. Jézéquel and J.-M. Chesneaux. CADNA: a library for estimating round-off error propagation. *Comput. Phys. Comm.* 178.12 (2008), 933–955.

[174] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Rev.* 51.3 (2009), 455–500.

[175] D. Kressner and C. Tobler. Algorithm 941: htucker—A MATLAB toolbox for tensors in hierarchical Tucker format. *ACM Trans. Math. Software* 40.3 (2014), 1–22.

[176] R. Kriemann. Hierarchical lowrank arithmetic with binary compression. arXiv:2308.10960. 2023.

[177] R. Kriemann. Performance of H-matrix-vector multiplication with floating point compression. arXiv:2405.03456. 2024.

[178] U. W. Kulisch and W. L. Miranker. The arithmetic of the digital computer: a new approach. *SIAM Rev.* 28.1 (1986), 1–40.

[179] J. Kurzak and J. Dongarra. Implementation of mixed precision in solving systems of linear equations on the Cell processor. *Concurr. Comput. Pract. Exp.* 19.10 (2007), 1371–1385.

[180] J.-Y. L'Excellent. Multifrontal methods: parallelism, memory usage and numerical aspects. PhD thesis. Ecole normale supérieure de Lyon, 2012.

[181] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. 2006, 50–50.

[182] Q.-T. Le, E. Riccietti, and R. Gribonval. Spurious valleys, NP-hardness, and tractability of sparse matrix factorization with fixed support. *SIAM J. Matrix Anal. Appl.* 44.2 (2023), 503–529.

[183] Q.-T. Le, L. Zheng, E. Riccietti, and R. Gribonval. Fast learning of fast transforms, with guarantees. *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2022, 3348–3352.

[184] Q.-T. Le, L. Zheng, E. Riccietti, and R. Gribonval. Butterfly factorization with error guarantees. arXiv:2411.04506. 2025.

[185] X. S. Li and J. W. Demmel. SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Software* 29.2 (2003), 110–140.

[186] X. S. Li and J. W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, Washington, DC, USA, 1998, 1–17.

[187] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proc. Nat. Acad. Sci.* 104.51 (2007), 20167–20172.

[188] N. Lindquist, P. Luszczek, and J. Dongarra. Improving the performance of the GMRES method using mixed-precision techniques. *Communications in Computer and Information Science*. Ed. by J. Nichols, B. Verastegui, A. 'B. Maccabe, O. Hernandez, S. Parete-Koon, and T. Ahearn. Springer, Cham, Switzerland, 2020, 51–66.

[189] N. Lindquist, P. Luszczek, and J. Dongarra. Replacing pivoting in distributed Gaussian elimination with randomized techniques. *2020 IEEE/ACM 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. 2020, 35–43.

[190] N. Lindquist, P. Luszczek, and J. Dongarra. Accelerating restarted GMRES with mixed precision arithmetic. *IEEE Trans. Parallel Distrib. Syst.* 33.4 (2022), 1027–1037.

[191] N. Lindquist, P. Luszczek, and J. Dongarra. Generalizing random butterfly transforms to arbitrary matrix sizes. *ACM Trans. Math. Software* 50.4 (Dec. 2024).

[192] Y. Liu, P. Ghysels, L. Claus, and X. S. Li. Sparse approximate multifrontal factorization with butterfly compression for high-frequency wave equations. *SIAM J. Sci. Comput.* 43.5 (2021), S367–S391.

[193] J. A. Loe, C. A. Glusa, I. Yamazaki, E. G. Boman, and S. Rajamanickam. Experimental evaluation of multiprecision strategies for GMRES on GPUs. *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, 469–478.

[194] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. NVIDIA tensor core programmability, performance & precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, 522–531.

[195] P.-G. Martinsson, G. Quintana Ortí, N. Heavner, and R. van de Geijn. Householder QR factorization with randomization for column pivoting (HQRRP). *SIAM J. Sci. Comput.* 39.2 (2017), C96–C115.

[196] P.-G. Martinsson and S. Voronin. A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices. *SIAM J. Sci. Comput.* 38.5 (2016), S485–S507.

[197] M. Melnichenko, O. Balabanov, R. Murray, J. Demmel, M. W. Mahoney, and P. Luszczek. CholeskyQR with randomization and pivoting for tall matrices (CQRRPT). arXiv:2311.08316. 2025.

[198] P. Micikevicius, S. Oberman, P. Dubey, M. Cornea, A. Rodriguez, I. Bratt, R. Grisenthwaite, N. Jouppi, C. Chou, A. Huffman, M. Schulte, R. Wittig, D. Jani, and S. Deng. OCP 8-bit floating point specification (OFP8). Version 1.0. June 2023.

[199] R. Minster, A. K. Saibaba, and M. E. Kilmer. Randomized algorithms for low-rank tensor decompositions in the Tucker format. *SIAM J. Math. Data Sci.* 2.1 (2020), 189–215.

[200] G. Moreau. On the solution phase of direct solvers for sparse linear systems with multiple sparse right-hand sides. PhD thesis. ENS Lyon ; Université de Lyon, Dec. 2018.

[201] R. B. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM J. Matrix Anal. Appl.* 16.4 (1995), 1154–1171.

[202] R. B. Morgan. GMRES with deflated restarting. *SIAM J. Sci. Comput.* 24.1 (2002), 20–37.

[203] D. Mukunoki, M. Kawai, and T. Imamura. Sparse matrix-vector multiplication with reduced-precision memory accessor. *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE. 2023, 608–615.

[204] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura. DGEMM using tensor cores, and its accurate and reproducible versions. *High Performance Computing*. Cham: Springer International Publishing, 2020, 230–248.

[205] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, et al. *Handbook of Floating-Point Arithmetic*. Springer, 2018.

[206] Y. Nagasaka, A. Nukada, and S. Matsuoka. Adaptive multi-level blocking optimization for sparse matrix vector multiplication on GPU. *Procedia Comput. Sci.* 80 (2016), 131–142.

[207] Y. Nakatsukasa and J. A. Tropp. Fast and accurate randomized algorithms for linear systems and eigenvalue problems. *SIAM J. Matrix Anal. Appl.* 45.2 (2024), 1183–1214.

[208] *NVIDIA Blackwell Architecture Technical Brief*. V1.0. NVIDIA, Mar. 2024.

[209] E. Oktay and E. Carson. Multistage mixed precision iterative refinement. *Numer. Linear Algebra Appl.* 29.4 (2022), e2434.

[210] H. Ootomo, K. Ozaki, and R. Yokota. DGEMM on integer matrix multiplication unit. *Int. J. High Perform. Comput. Appl.* 38.4 (2024), 297–313.

[211] H. Ootomo and R. Yokota. Recovering single precision accuracy from tensor cores while surpassing the fp32 theoretical peak performance. *Int. J. High Perform. Comput. Appl.* 36.4 (2022), 475–491.

[212] H. Ootomo and R. Yokota. Mixed-precision random projection for RandNLA on tensor cores. *Proceedings of the Platform for Advanced Scientific Computing Conference*. Davos, Switzerland: Association for Computing Machinery, 2023.

[213] S. Operto, A. Miniussi, R. Brossier, L. Combe, L. Métivier, V. Monteiller, A. Ribodetti, and J. Virieux. Efficient 3-D frequency-domain mono-parameter full-waveform inversion of ocean-bottom cable data: application to Valhall in the visco-acoustic vertical transverse isotropic approximation. *Geophys. J. Int.* 202.2 (2015), 1362–1391.

[214] R. Orús. A practical introduction to tensor networks: matrix product states and projected entangled pair states. *Ann. Phys.* 349 (2014), 117–158.

[215] I. V. Oseledets. Tensor-train decomposition. *SIAM J. Sci. Comput.* 33.5 (2011), 2295–2317.

[216] I. V. Oseledets and E. E. Tyrtyshnikov. Breaking the curse of dimensionality, or how to use SVD in many dimensions. *SIAM J. Sci. Comput.* 31.5 (2009), 3744–3759.

[217] K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump. Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. *Numer. Algorithms* 59 (2012), 95–118.

[218] K. Ozaki, Y. Uchino, and T. Imamura. Ozaki scheme II: a GEMM-oriented emulation of floating-point matrix multiplication using an integer modular technique. arXiv:2504.08009. 2025.

[219] C. C. Paige, M. Rozložník, and Z. Strakoš. Modified Gram–Schmidt (MGS), least squares, and backward stability of MGS-GMRES. *SIAM J. Matrix Anal. Appl.* 28.1 (2006), 264–284.

[220] D. S. Parker. *Random butterfly transformations with applications in computational linear algebra*. UCLA Computer Science Department, 1995.

[221] M. L. Parks, E. de Sturler, G. Mackey, D. D. Johnson, and S. Maiti. Recycling Krylov subspaces for sequences of linear systems. *SIAM J. Sci. Comput.* 28.5 (2006), 1651–1674.

[222] J. Peca-Medlin and T. Trogdon. Growth factors of random butterfly matrices and the stability of avoiding pivoting. *SIAM J. Matrix Anal. Appl.* 44.3 (2023), 945–970.

[223] C. Pernet and A. Storjohann. Time and space efficient generators for quasiseparable matrices. *J. Symb. Comput.* 85 (2018), 224–246.

[224] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman. Sparse supernodal solver using block low-rank compression: design, performance and analysis. *J. Comput. Sci.* 27 (2018), 255–270.

[225] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Math. Software* 42.4 (June 2016), 27:1–27:35.

[226] B. D. Rouhani, N. Garegrat, T. Savell, A. More, K.-N. Han, R. Zhao, M. Hall, J. Klar, E. Chung, Y. Yu, M. Schulte, R. Wittig, I. Bratt, N. Stephens, J. Milanovic, J. Brothers, P. Dubey, M. Cornea, A. Heinecke, A. Rodriguez, M. Langhammer, S. Deng, M. Naumov, P. Micikevicius, M. Siu, and C. Verrilli. OCP microscaling formats (MX) specification. Version 1.0. Sept. 2023.

[227] S. M. Rump. Ultimately fast accurate summation. *SIAM J. Sci. Comput.* 31.5 (2009), 3466–3502.

[228] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput.* 31.1 (2008), 189–224.

[229] S. M. Rump, T. Ogita, and S. Oishi. Fast high precision summation. *Nonlinear Theory and Its Applications, IEICE* 1.1 (2010), 2–24.

[230] Y. Saad. A flexible inner-outer preconditioned GMRES algorithm. *SIAM J. Sci. Comput.* 14.2 (1993), 461–469.

[231] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.* 7.3 (1986), 856–869.

[232] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics, 2003.

[233] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker. PARDISO: A high-performance serial and parallel sparse linear solver in semiconductor device simulation. *Future Gener. Comput. Syst* 18.1 (2001), 69–78.

[234] V. Simoncini and D. B. Szyld. Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM J. Sci. Comput.* 25.2 (2003), 454–477.

[235] R. D. Skeel. Iterative refinement implies numerical stability for Gaussian elimination. *Math. Comput.* 35.151 (1980), 817–832.

[236] G. L. G. Sleijpen and H. A. van der Vorst. Reliable updated residuals in hybrid Bi-CG methods. *Computing* 56.2 (June 1996), 141–163.

[237] A. Sorna, X. Cheng, E. D'Azevedo, K. Won, and S. Tomov. Optimizing the fast Fourier transform using mixed precision on tensor core hardware. *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*. 2018, 3–7.

[238] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM J. Sci. Comput.* 23.6 (2002), 2165–2182.

[239] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.* 13.4 (1969), 354–356.

[240] T. Terao, K. Ozaki, and T. Ogita. LU-Cholesky QR algorithms for thin QR decomposition. *Parallel Computing* 92 (2020), 102571.

[241] K. Turner and H. F. Walker. Efficient high accuracy solutions with GMRES(m). *SIAM J. Sci. Statist. Comput.* 12.3 (1992), 815–825.

[242] Y. Uchino, K. Ozaki, and T. Imamura. Performance enhancement of the Ozaki scheme on integer matrix multiplication unit. *Int. J. High Perform. Comput. Appl.* 39.3 (2025), 462–476.

[243] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen. A new truncation strategy for the higher-order singular value decomposition. *SIAM J. Sci. Comput.* 34.2 (2012), A1027–A1052.

[244] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017.

[245] B. Vieublé. Mixed precision iterative refinement for the solution of large sparse linear systems. PhD thesis. INP Toulouse, Nov. 2022.

[246] J. A. Vogel. Flexible BiCG and flexible Bi-CGSTAB for nonsymmetric linear systems. *Appl. Math. Comput.* 188.1 (May 2007), 226–233.

[247] H. A. van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.* 13.2 (Mar. 1992), 631–644.

[248] N. Wang, J. Choi, D. Brand, C. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, 7686–7695.

[249] J. H. Wilkinson. Progress report on the automatic computing engine. 1948.

[250] J. H. Wilkinson. Error analysis of direct methods of matrix inversion. *J. Assoc. Comput. Mach.* 8 (1961), 281–330.

[251] J. H. Wilkinson. *Rounding Errors in Algebraic Processes*. London: Notes on Applied Science No. 32, Her Majesty's Stationery Office, 1963, vi+161.

[252] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numer. Linear Algebra Appl.* 17.6 (2010), 953–976.

[253] J. Xiao, M. Gu, and J. Langou. Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations. *2017 IEEE 24th International Conference on High Performance Computing (HiPC)* (2017), 233–242.

[254] Y. Xu, L. Xie, X. Gu, X. Chen, H. Chang, H. Zhang, Z. Chen, X. Zhang, and Q. Tian. QA-LoRA: quantization-aware low-rank adaptation of large language models. arXiv:2309.14717. 2023.

[255] I. Yamazaki, H. Anzt, S. Tomov, M. Hoemmen, and J. Dongarra. Improving the performance of CA-GMRES on multicores with multiple GPUs. *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, 382–391.

[256] I. Yamazaki, S. Tomov, and J. Dongarra. Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. *SIAM J. Sci. Comput.* 37.3 (2015), C307–C330.

[257] I. Yamazaki, S. Tomov, and J. Dongarra. Stability and performance of various singular value QR implementations on multicore CPU with a GPU. *ACM Trans. Math. Software* 43.2 (Sept. 2016).

[258] Y. Zhao, T. Fukaya, and T. Iwashita. Numerical behavior of mixed precision iterative refinement using the BiCGStab method. *J. Inf. Process.* 31 (2023), 860–874.

[259] Y. Zhao, T. Fukaya, L. Zhang, and T. Iwashita. Numerical investigation into the mixed precision GMRES(m) method using FP64 and FP32. *J. Inf. Process.* 30 (2022), 525–537.

[260] L. Zheng, E. Riccietti, and R. Gribonval. Efficient identification of butterfly sparse matrix factorizations. *SIAM J. Math. Data Sci.* 5.1 (2023), 22–49.

[261] Y.-K. Zhu and W. B. Hayes. Algorithm 908: online exact summation of floating-point streams. *ACM Trans. Math. Software* 37.3 (2010), 1–13.

[262] Y.-K. Zhu and W. B. Hayes. Correct rounding and hybrid approach to exact floating-point summation. *SIAM J. Sci. Comput.* 31.4 (2009), 2981–3001.

[263] Z. Zlatev. Use of iterative refinement in the solution of sparse linear systems. *SIAM J. Numer. Anal.* 19.2 (1982), 381–399.

[264] Q. Zou. Probabilistic rounding error analysis of modified Gram–Schmidt. *SIAM J. Matrix Anal. Appl.* 45.2 (2024), 1076–1088.